Open Multithreaded Transactions

A Transaction Model for Concurrent Object-Oriented Programming

THÈSE N^{o} 2393 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jörg Kienzle

Ingénieur Informaticien Diplômé EPF

Originaire de Hofstetten-Flüh (SO)

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse

Prof. Rachid Guerraoui, rapporteur

Prof. Rudolf Keller, rapporteur

Prof. Oscar Nierstrasz, rapporteur

Dr. Alexander Romanovsky, rapporteur

Lausanne, EPFL

Avril 2001

Abstract

Modern programming languages provide features that allow a programmer to express concurrency in an application by using active objects, i.e. objects with their own thread of control, and distribution. Concurrent systems can be classified into *cooperative* systems, where individual components collaborate, share results and work for a common goal, and *competitive* systems, where the individual components are not aware of each other and compete for shared resources. Programming languages address collaboration and competition by providing means for communication and synchronization among active objects.

The realization of complex object-oriented systems often needs sophisticated and elaborate concurrency features which may go beyond the traditional concurrency control associated with separate method calls. A *transaction* groups together a sequence of actions, and can therefore encapsulate complex behavior and embrace groups of objects and method calls. Transactions structure the dynamic system execution as opposed to the static structuring based on objects. Because of the ACID properties, transactions are able to hide the effects of concurrency and at the same time act as firewalls for errors, making them appropriate building blocks for structuring reliable distributed systems. This thesis investigates how transactions can be integrated with concurrent object-oriented programming, and in particular, how transactions can be made available to an application programmer at the programming language level.

In the first part of the thesis, existing transaction models are reviewed and their suitability for concurrent programming languages is discussed. The analysis of existing models of multithreaded transactions shows that they either give too much freedom to threads and do not control their participation in transactions, or unnecessarily restrict the computational model by assuming that only one thread can enter a transaction. Hence, a significant part of this thesis is devoted to the establishment of a new transaction model named *Open Multithreaded Transactions*, providing features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior in order to guarantee correctness of transaction nesting and isolation among transactions.

The open multithreaded transaction model incorporates disciplined exception handling adapted to nested transactions. It allows individual threads to perform *forward error recovery* by handling an abnormal situation locally, and promotes a defensive approach for developing transactional objects, so that errors are detected early and dealt with inside the transaction. If local handling fails, the transaction support applies *backward error recovery* and reverses the system to its “initial” state.

The second part of the thesis describes the design of an object-oriented framework called OPTIMA, which provides the necessary run-time support for open multithreaded transactions. Since applications from many different domains can benefit from using transactions, it is important to allow an application programmer to customize the framework. This flexibility is achieved with the help of design patterns. Class hierarchies with classes implementing standard transactional behavior are provided, but a programmer is free to extend the hierarchies to tailor the framework to the application-specific needs. The framework supports among others optimistic and pessimistic concurrency control, different recovery strategies (i.e. Undo/Redo, NoUndo/Redo, Undo/NoRedo), different caching techniques, different logging techniques (i.e. physical logging and logical logging), and different storage devices. Among pessimistic concurrency control, the framework provides built-in support for lock-based concurrency control, with strict read / write or commutativity-based locking.

An important decision was to provide support for open multithreaded transactions in a programming language without modifying the language itself, which avoids having to modify the compiler. Possible interfaces for an application programmer are exposed, including a procedural, an object-based and an object-oriented interface. The feasibility and the elegance of the interfaces depend on the available features of the programming language.

The third part of the thesis presents an implementation of the OPTIMA framework for the concurrent object-oriented programming language Ada 95. It has been realized in form of a library based on standard Ada only. This makes the approach useful for all settings and platforms which have standard Ada compilers. Based on the features offered by Ada 95, a procedural, an object-based and an object-oriented interfaces for the transaction framework have been implemented. The prototype implementation validates the appropriateness of the design of the framework, and allows application programmers to experiment with the open multithreaded transaction model.

The last part of the thesis describes the design and implementation of an auction system based on open multithreaded transactions. This case study, an example of a dynamic system with cooperative and competitive concurrency, validates the open multithreaded transaction model. It shows how the complexity of the application can be reduced by structuring the execution using open multithreaded transactions. Reasoning about fault tolerance issues and consistency of the overall system is made a lot easier. Due to the isolation property and disciplined exception handling, open multithreaded transactions do not allow errors to propagate to the outside, and therefore constitute units of fault tolerance.

Résumé

Les langages de programmation modernes comprennent des mécanismes permettant d'exprimer la concurrence inhérente à une application à travers les objets actifs, c'est-à-dire des objets qui ont leur propre fil d'exécution (tâche), et la répartition. Il existe deux types de systèmes concurrents: les systèmes *coopérants*, où les composants individuels collaborent, partagent leurs résultats et travaillent dans un même but, et les systèmes *compétitifs*, où les composants individuels s'ignorent les uns les autres et se disputent les ressources partagées. Les langages de programmation abordent les problèmes de collaboration et de compétition en offrant des mécanismes de communication et de synchronisation parmi les objets actifs.

Lorsque l'on réalise des systèmes à objets complexes, il est souvent nécessaire de disposer de mécanismes de contrôle de concurrence plus sophistiqués et plus évolués que ceux traditionellement associés avec les appels de méthodes individuelles. Une *transaction* regroupe une séquence d'action et peut ainsi renfermer un comportement complexe et englober des groupes d'objets et d'appels aux méthodes. Les transactions structurent l'exécution dynamique d'un système, par opposition à la structure statique réalisée par la décomposition en objets. Grâce aux propriétés ACID, les transactions permettent de cacher les problèmes liés à la concurrence et empêchent en même temps la propagation d'erreurs. Celles-ci forment ainsi des blocs d'éléments de base appropriés à la structuration de systèmes répartis fiables. Cette thèse étudie la manière dont les transactions peuvent être intégrées à la programmation à objets concurrente et examine en particulier comment les transactions peuvent être mises à disposition au niveau du langage de programmation.

La première partie de la thèse décrit les modèles de transaction existants tout en examinant leur place éventuelle dans un langage de programmation concurrent. L'analyse des modèles de transaction multitâches montre que soit les tâches ont trop de liberté, leur participation à la transaction n'étant pas contrôlée, soit le modèle est trop restrictif en ne laissant qu'une seule tâche entrer dans une transaction. Pour cette raison, une partie importante de la thèse a été consacrée à l'établissement d'un nouveau modèle appelé transactions multitâches ouvertes (*Open Multithreaded Transactions*). Le modèle offre non seulement les mécanismes habituels qui permettent de contrôler et de structurer l'accès aux objets, mais aussi la possibilité de superviser les tâches qui participent à la transaction. Plusieurs tâches ont le droit de pénétrer dans une même transaction pour travailler en commun. Ce modèle autorise également la création de nouvelles tâches de même que leur destruction à l'intérieur ou à l'extérieur d'une transaction. Ce comportement est limité à certains endroits pour obtenir une imbrication correcte et pour garantir l'isolation entre les transactions.

Le modèle présenté intègre également un traitement d'exceptions structuré. Les tâches participant à une transaction peuvent appliquer une politique de *recouvrement en avant* et donc essayer de rétablir une situation correcte localement. Une approche défensive pour le développement d'objets transactionnels est également préconisée, afin de détecter

les erreurs au plus tôt et de permettre leur traitement à l'intérieur même de la transaction. Si les démarches locales échouent, le support transactionnel applique le *recouvrement en arrière* et remet le système dans son état initial.

La deuxième partie de la thèse décrit la conception d'un framework (cadre applicatif à objets) baptisé OPTIMA, qui offre le support nécessaire pour les transactions multitâches ouvertes à l'exécution. Etant donné que des applications dans de nombreux domaines différents peuvent bénéficier de l'utilisation de transactions, il est important de permettre à un programmeur de modifier le framework à la demande. Cette flexibilité est atteinte à l'aide de "design patterns". Une hiérarchie de classes est mise à disposition, et des classes concrètes implémentent plusieurs comportements transactionnels standards. Si nécessaire, un programmeur peut étendre la hiérarchie pour adapter le framework aux exigences de son application. Ce framework offre parmi d'autres un contrôle de concurrence optimiste et pessimiste, plusieurs stratégies de reprise après défaillance (Défaire / Refaire, Ne Pas Défaire / Refaire, Défaire / Ne Pas Refaire), différentes techniques de gestion de cache, la journalisation physique et logique, et permet de gérer des unités de stockage de différents types. Parmi les méthodes de contrôle de concurrence pessimiste, le framework met à disposition le verrouillage à deux phases strict ou typé, basé sur la commutativité des opérations de l'objet transactionnel en question.

Le choix d'intégrer les transactions multitâches ouvertes dans un langage de programmation sans modifier le langage lui-même rend le framework plus portable et maintenable. Le programmeur peut bénéficier de différentes interfaces, notamment d'une interface procédurale, d'une interface s'appuyant sur des objets (object-based), et d'une interface à objets. L'éventuelle réalisation des interfaces et leur mise en œuvre dépendent des mécanismes mis à disposition par le langage de programmation en question.

La troisième partie de la thèse expose une implémentation du framework OPTIMA pour le langage de programmation à objets concurrent Ada 95. La réalisation est écrite en Ada standard sous forme de bibliothèque, et est ainsi utilisable sur toutes les plateformes qui proposent un compilateur Ada standard. Les interfaces mentionnées précédemment ont été réalisées en s'appuyant sur les mécanismes de concurrence offerts par Ada 95. Cette implémentation valide les concepts du framework et permet à un programmeur de mettre en pratique les transactions multitâches ouvertes.

La dernière partie de la thèse décrit la conception et la réalisation d'un système de vente aux enchères électronique utilisant les transactions multitâches ouvertes. Cette étude de cas, qui présente simultanément des traits de concurrence coopérative et compétitive, valide le modèle et montre également comment la complexité de l'application peut être réduite en structurant l'exécution par les transactions multitâches ouvertes. De plus, ce procédé facilite la gestion de la tolérance aux pannes et garantit la cohérence globale du système. La propriété d'isolation empêche la propagation éventuelle d'erreurs à l'extérieur de la transaction. Les transactions multitâches ouvertes peuvent donc être considérées comme les unités de tolérance aux défaillances.

Acknowledgements

This thesis, as it stands here, could not have been completed without the help of others. I would like to express my gratitude to them.

First of all, I would like to thank Professor Alfred Strohmeier for supervising this research work and for supporting me through all these years. He has given me a lot of freedom by letting me choose the topic of my Ph.D., and has given me ample opportunities to meet some of the experts in the field by sending me to renowned conferences. He has always found time for me, and his accuracy and his talent for pointing out the essential have helped me to bring out the best in my papers and in my research work in general. I also thank him for offering me the opportunity to teach, and for having read draft versions of this thesis; I think it has benefited greatly from his criticisms.

I am grateful to the jury members Professor Rachid Guerraoui, Professor Rudolf Keller, Professor Oscar Nierstrasz, and Dr. Alexander Romanovsky for having accepted to serve on my examination board and for the time they invested to read and evaluate this work. I am indebted to Dr. Alexander Romanovsky for the intense discussions we had during his visit at the Software Engineering Lab. His knowledge and experience in the field of fault tolerance have been invaluable for me. His advice and suggestions helped me give shape to the open multithreaded transaction model, and he pushed me to work out the interaction with exception handling in detail. His enthusiastic emails have been very stimulating, and I very much appreciate his openness and sympathy.

I also want to express my thanks to Dr. Ricardo Jiménez-Peris for allowing me to start the work on the OPTIMA framework on his bases, and for assisting me throughout the whole development.

I am grateful to Professor Andy Wellings, Professor Bo Sanden, Dr. Bob Johnson, Dr. Thomas Wolf and Stephen Michell for the interesting research we performed together on extensible protected types. This far-reaching work has given me deep insight into concurrent object-oriented programming languages and Ada in particular.

Special thanks go to Dr. Thomas Wolf, my “buddy” and former office mate, on whom I can always count on when I am in “scientific distress”. I thank him for telling me the ways of Ada 95 and for his sincere friendship. I am grateful to Shane Sendall for his careful review of this thesis, and to Xavier Caron for the work he accomplished in his diploma thesis on stable storage and for implementing parts of the auction system. I also thank the other members of the Software Engineering Lab, Anne Schlageter, Dr. Didier Buchs, Mohamed Kandé, Benjamin Barras, Enzo Grigio, Stanislav Chachkov, Adel Besrour, David Hürzeler, Sandro Costa, Raul Silaghi and Rodrigo Garcia-Garcia for the pleasant atmosphere.

I thank the students of the first year programming class in the years 2000 and 2001 for making teaching such an enjoyable task, and in particular Simon Schule for implementing parts of the graphical user interface of the auction system.

I am grateful to my parents, and especially to my father, for always supporting my different undertakings, whether they were related to computers or to ice skating. I also want to thank the parents of my wife for welcoming me in their family and for the many philosophical discussions. I will never forget my ice dancing coach Walter Hofer, a man of good advice, who has taught me close teamwork and how to last out.

I also want to thank my “Gemini Brother” Yann Le Tensorer and Henrik Gudat for all the fun we had in the past developing computer games and writing songs.

Finally, I thank my daughters Julie, Laura and Isabelle for their smiles and their positive energy, and for periodically reminding me of how much work is left to do:

Laura: “Hi Daddy! How was work today? Did you finish your thesis?”
Me: “Well, only half of chapter 7.”
Julie: “Great, that leaves you with only 6 $\frac{1}{2}$ chapters to write, plus the introduction and conclusion.”

But most of all I thank my wife Valérie for her cheerfulness and great sense of humor. She makes life worth living, and a thesis worth writing! Without her support I might not have accomplished this work. She watched with devotion over our three daughters, and raised my spirits more than once when stress got the better of me.

To Valérie

Table of Contents

List of Figures

Part I: Transaction Models

Chapter 1: Fundamental Concepts

Chapter 2: Transaction Models

Chapter 3: Open Multithreaded Transactions

Part II: The OPTIMA Framework

Chapter 4: Overall Design

Chapter 5: Transaction Support

Chapter 6: Concurrency Control

Chapter 7: Recovery

Chapter 8: Interfacing with Programming Languages

Part III: Implementation for Ada 95

Chapter 9: Ada 95

Chapter 10: Implementation for Ada 95

Chapter 11: Related Work

Part IV: Case Study

Chapter 12: Online Auction System

Chapter 13: Conclusion

Part V: Annexes

Bibliography

Author and Citation Index

Curriculum Vitae

Chapter 1:
Introduction

1.1 Context and Objectives

Computer software has become a driving technology. It is embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial processes, entertainment, appliances, office products, etc. Software is virtually inescapable in a modern world. People have accepted the omnipresence of software, and view it as a technological fact of life.

As a consequence, applications must respond to an increasing amount of demands. To satisfy user expectations, applications offer more and more functionality, and hence grow more complex. Fancy user interfaces or interaction with real-time devices, e.g. sensors, require software to promptly respond to external stimuli and to be able to perform several operations simultaneously. There is also an increasing need for integrating different systems and applications, which results in heterogeneous and possibly distributed systems. Moreover, the ever increasing popularity of the Internet and the growing field of e-commerce have lead to an explosion of the number of distributed systems in operation. Such systems typically are required to provide highly available services, and must therefore satisfy hundreds of clients simultaneously. For the same reason, fault tolerance requirements, that used to be applied only in the domain of mission-critical and safety-critical systems, are nowadays also applied to other distributed systems.

Modern programming languages such as Java [GJS96] or Ada [ISO95] reflect this trend. They provide features that allow a programmer to express concurrency and distribution in the program code, and provide exceptions as a standard way for handling abnormal situations. Objects support concurrency, resulting in so-called active objects, i.e. objects

with their own thread of control, and can be distributed by assigning them to physical nodes in a network.

Many researchers view all object-oriented systems as inherently concurrent, since objects themselves are “naturally concurrent” entities. In reality, concurrency adds a new dimension to system structure and design. Concurrent systems are extremely difficult to understand, design, analyze and modify.

In general, the execution of a sequential object-oriented program starts by executing a method of a certain object, which in turn calls methods on other objects, etc. At any given time, only one object is executing one of its methods. In concurrent object-oriented programs, several methods can be active at a given time, and even a given method might be invoked multiple times concurrently. Objects must be aware of this concurrency in order to guarantee state consistency in the presence of simultaneous method invocations.

Concurrent systems can be classified into *cooperative* systems, where individual components collaborate, share results and work for a common goal, and *competitive* systems, where the individual components are not aware of each other and compete for shared resources. In general, concurrent programming languages address collaboration and competition by providing means for communication and synchronization among active objects.

However, concurrent and distributed computing often gives rise to complex concurrent and interacting activities. Sophisticated object-oriented systems often need more advanced and elaborate concurrency features which may go beyond the traditional concurrency control associated with separate method calls. Because multiple objects must usually be accessed or updated together to correctly reflect the real world, great care must be taken to keep related objects globally consistent. Any interruption of updates to objects, or the interleaving of updates and accesses, can break the overall consistency of an object system.

In the field of database systems, the notion of *transaction* has been introduced to solve a similar problem, i.e. to correctly handle interrelated and concurrent updates of data and to provide fault tolerance with respect to hardware failures. But very soon, transactions have been applied to a wider range of domains, especially to distributed systems. Jim Gray, one of the experts of transactions, has said in the foreword for [Elm93]:

> *The transaction concept has emerged as the key structuring technique for distributed data and distributed computations. Originally developed and applied to database applications, the transaction model is now being used in new application areas ranging from process control to cooperative work. Not surprisingly, these more sophisticated applications require a refined and generalized transaction model. The concept must be made recursive, it must deal with concurrency within a transaction, it must relax the strict isolation among transactions, and it must deal more gracefully with failures.*
>
> *Jim Gray*

A transaction groups together a sequence of actions. It can therefore encapsulate complex behavior and embrace groups of objects and method calls. Transactions structure dynamic system execution as opposed to static structuring based on objects.

Transactions are programmed in a way that they move an application from one consistent state to another. Transaction processing technology ensures that each transaction is either executed to completion or not at all, and that concurrently executed transactions behave as though each transaction executes in isolation. Moreover, once a transaction completes successfully, its changes to the states of objects are made persistent. These properties are called the ACID properties: atomicity, consistency, isolation and durability. The significance of transactions is magnified by the fact that these guarantees are upheld despite failures of computer components, distribution of data across multiple computers, and overlapping or parallel execution of different transactions.

If problems are encountered during the execution of a transaction, the transaction can be aborted, which results in undoing all changes that have been made to transactional objects on behalf of the transaction so far. Thanks to the isolation property, errors that appear inside a transaction can not spread to other parts of the application. Aborting a transaction therefore results in applying backward error recovery: consistency of the application state is restored.

Transactions are a very powerful concept, since they are able to hide the effects of concurrency, and therefore reduce complexity, and at the same time act as firewalls for errors, making them appropriate building blocks for structuring reliable, concurrent, and possibly distributed systems.

This thesis investigates how transactions can be integrated with concurrent object-oriented programming, and in particular, how transactions can be provided to an application programmer at the programming language level.

1.2 Contributions of this Thesis

The main contributions of this thesis are the following:

- A new transaction model called *Open Multithreaded Transactions*, suitable for concurrent object-oriented programming.

  The open multithreaded transaction model addresses *cooperative* and *competitive* concurrency as found in modern programming languages. It provides features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a common activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior when necessary in order to guarantee

correctness of transaction nesting and isolation among transactions. The model incorporates disciplined exception handling adapted to nested transactions. It allows individual threads to perform *forward error recovery* by handling an abnormal situation locally, and promotes a defensive approach for developing transactional objects, so that errors are detected early and dealt with inside the transaction. If local handling fails, the transaction support applies *backward error recovery* and reverses the system to a previous consistent state.

- An object-oriented framework called *OPTIMA*, that provides support for transactions in general, and in particular for open multithreaded transactions.

  The framework is based on design patterns, making it possible to customize and extend the framework according to the application needs. Class hierarchies with classes implementing standard transactional behavior are provided, but a programmer is free to extend the hierarchies to implement application-specific transaction control. The framework provides:

  – Support for optimistic and pessimistic concurrency control
  – Support for strict and semantic-based concurrency control
  – Undo/Redo, NoUndo/Redo and Undo/NoRedo recovery strategies
  – Customizable caching techniques
  – Support for in-place and deferred update
  – Support for physical and logical logging
  – Customizable support for storage.

- A *procedural interface*, an *object-based interface*, and an *object-oriented interface* that allow an application programmer to interact with the OPTIMA framework.

  Only classic procedural and object-oriented programming techniques have been used to build these interfaces to the framework, and therefore the programming language itself does not need to be modified to take advantage of transactions. No compiler modifications are necessary.

- An *implementation* of the OPTIMA framework for the concurrent object-oriented programming language Ada.

  The implementation is based on standard Ada only, which makes it usable with any settings and platforms that provide standard Ada compilers.

- A *case study* that validates the concepts of the open multithreaded transaction model.

  The design and implementation of an auction system, an example of a dynamic system with cooperative and competitive concurrency, shows how complex, distributed,

fault-tolerant systems can be designed and structured using open multithreaded transactions.

1.3 Thesis Organization

The thesis is split into four main parts. Part I (chapter 2 - chapter 4) covers the theoretical part of the thesis, which consists in finding an appropriate transaction model that supports different forms of concurrency. Chapter 2 reviews the fundamental concepts found in modern programming languages that must be taken into account when looking for suitable transaction models. Object-orientation, concurrency, fault-tolerance, exceptions and persistence are the major topics of this chapter. From a historical perspective, two different kinds of atomic units dealing with concurrency have emerged: transactions, which emphasize competitive concurrency, and conversations, which emphasize cooperative concurrency. Both models have been extended in many ways, and recently, some of them even address both forms of concurrency. Chapter 3 presents an overview of this evolution. Unfortunately, none of the existing models offers all desired features and hence, chapter 4 defines a new transaction model named *Open Multithreaded Transactions*, integrating thread control and exception handling.

Part II (chapter 5 - chapter 9) presents the *OPTIMA* framework, an object-oriented framework providing support for open multithreaded transactions. Chapter 5 gives an overview of the design of the framework, which is split into three main components, namely *transaction support*, *concurrency control* and *recovery support*. The design patterns that have been used throughout the framework to achieve customizability and extensibility are presented. The detailed design description starts with chapter 6, in which the transaction support component is described. Responsible for the life cycle of an open multithreaded transaction, the transaction support also controls and synchronizes the participating threads. Chapter 7 presents the concurrency control component, which provides support for optimistic and pessimistic concurrency control strategies. It is shown how semantic knowledge of operations can be exploited to increase concurrency. The design of the recovery support component is exposed in chapter 8. The recovery manager, which implements the different recovery strategies, controls the interaction between the cache manager, the log, and the persistence support. Finally, chapter 9 shows how these three components work together, and presents different interfaces to the OPTIMA framework for application programmers. A procedural, an object-based, and an object-oriented interface are laid out in detail.

Part III (chapter 10 - chapter 12) covers the implementation of the OPTIMA framework for the concurrent object-oriented programming language Ada 95. In chapter 10, some of the advanced features of Ada, in particular those addressing concurrency, are reviewed. Chapter 11 shows how these features have been used in the implementation of the OPTIMA framework for Ada, and how they have determined the concrete form of the procedural, the

object-based, and the object-oriented interface that are provided to the application programmer. Chapter 12 finally reviews other transactional system implementations and their associated transaction models, as well as two middlewares offering transaction services.

Part IV (chapter 13) presents the design and implementation of an *auction system*, a typical distributed application found in the field of e-commerce. In light of this case study it is shown how a design based on open multithreaded transactions can reduce complexity resulting from cooperative and competitive concurrency, and at the same time achieve the required fault tolerance. Sample code taken from the Ada implementation of the auction system illustrates the use of the object-based Ada interface presented in chapter 11.

Finally, chapter 14 gives a conclusion, summarizing the main results of this work and indicating several directions for future research.

Part I

Transaction Models

Chapter 2:
Fundamental Concepts

This chapter reviews some fundamental concepts of concurrent object-oriented systems and the features found in programming languages that support them. The individual sections contain considerations that must be taken into account when designing a transaction model to be used in concurrent object-oriented programming languages. Note that the sections in this chapter are only intended to set the context for the discussions of the following chapters. They should not be considered complete summaries.

The main topics addressed in this chapter are object-orientation, concurrency, fault-tolerance, exceptions, and persistence.

2.1 Object-Orientation

Object-orientation is a way of thinking about problems. It is an approach to viewing the world and building software in terms of objects. Object-orientation is built upon well established principles, namely abstraction, information hiding, modularity and classification.

2.1.1 Base Principles

Abstraction
An *abstraction* of an entity focusses on the essential, inherent characteristics of the entity and ignores its accidental properties. The abstraction defines a contract upon which clients may depend.

Information Hiding
Information hiding is the process of hiding all the details of an entity that do not contribute to its essential characteristics. By using *encapsulation* one can separate the external aspects of an entity, i.e. its *interface*, from the internal implementation details. Communicating via the interface is the only way for the client to work with the entity.

Modularity
Modularization is the action of dividing a system into smaller pieces, i.e. modules, which can be developed separately. These modules form the structural architecture of the system. A module should be *cohesive*: it groups together logically related abstractions. Modules can be connected with other modules, but this coupling should be loose. Dependencies and interconnections among modules should be as few as possible. Providing clean modularity relies on encapsulation. Interfaces should be as narrow as possible.

Classification
Classification allows reasoning about entities based on kinds, sorts, or types. *Classes* group together entities marked by common properties: the similarities among the entities are promoted, the differences are ignored. This approach makes it possible to characterize by a finite set of properties an infinite set of possible entities. Classification is often hierarchical.

2.1.2 Concepts

The previously described principles of object-orientation are achieved using the notions of *objects* and *classes*.

An *object* represents an individual identifiable item, unit, or entity, either real or conceptual, with a well-defined role in the problem domain or in a system. When an object models a real-world entity, it is an abstraction of this entity. What is essential and what is accidental will depend on the application and on the developer. A *property* is an inherent or distinctive characteristic, trait, quality, or feature of an object.

An object has an *identity*, *state*, and *behavior*. Identity is the property that distinguishes an object from all other objects. It makes the object unique. The state of an object is its memory. The behavior of an object defines how it acts on its own initiative and how it reacts to external stimuli. The behavior of an object usually depends on its current state.

A class groups objects in such a way that the similarities can be promoted, and the differences ignored. Whereas an object is a concrete entity that exists in time and space, a class represents only the properties, the “essence” of an object, as it were. A class can be made of all the objects having the same internal structure, or a similar internal structure, and the same behavior, or a similar behavior.

The second view of the class concept is that it can be seen as a template or mold from which objects can be instantiated, i.e. created. In that case the created object is said to be an *instance* of the corresponding class. This feature gives classes an economic interest: instead

of describing the properties and behavior of each object individually, it suffices to describe their classes.

Object-orientation can be used throughout all the phases of software development [Mey97]. Software analysis, architecture, design and implementation can use object-orientation. Several object-oriented software development methods have been developed. Popular methods include OMT [RBP+91], Booch [Boo94], and Fusion [CAB+94].

The *Unified Modeling Language* [RJB99], UML for short, is a graphical modelling language that provides the means for describing object-oriented systems. It began in 1994 as an attempt to unify the Booch and OMT models, but quickly developed into a broadly based effort to standardize object-oriented modeling concepts, terminology, and notation. In 1997, UML was adopted as a standard by the Object Management Group, OMG for short, which also has the responsibility for future evolution of the UML. The UML specification contains a meta-model of modeling constructs, constraints defining well-formedness of models, definitions of semantics of the constructs, and notation for expressing models visually. UML does not standardize the development process, but is intended to support many current and future processes.

The UML notation will be used in this thesis whenever possible, especially in the figures on the design of the transaction framework presented in chapter 5 to chapter 9.

2.1.3 Object-Oriented Programming

Object-oriented programming is a method of programming inspired by the principles of object-orientation and is based on the object-oriented concepts mentioned above. Object-oriented programming languages encapsulate data *as well as* operations applicable to that data into objects.

2.1.4 Evolution of Object-Oriented Programming

Early programmers thought of programs as instruction sequences. Procedure-oriented languages introduced procedural abstractions that encapsulate sequences of actions into procedures. The procedure-oriented paradigm has strong organizing principles for managing actions and algorithms, but has weak organizing principles for managing shared data.

Typed languages introduced the notion of a *data type*. A type characterizes a set of values, and a set of operations applicable to those values. By adding means for providing encapsulation, the notion of type has been extended to be close to what we nowadays call object. At that time, objects providing a clearly defined interface and hiding the internal implementation were called *abstract data types*, ADT for short.

A programming language is said to be *object-based* if it supports objects as a language feature, and is said to be *object-oriented* if, additionally, objects are required to belong to classes that can be incrementally modified through inheritance [Weg90].

The evolution of programming language concepts is summarized in figure 2.1.

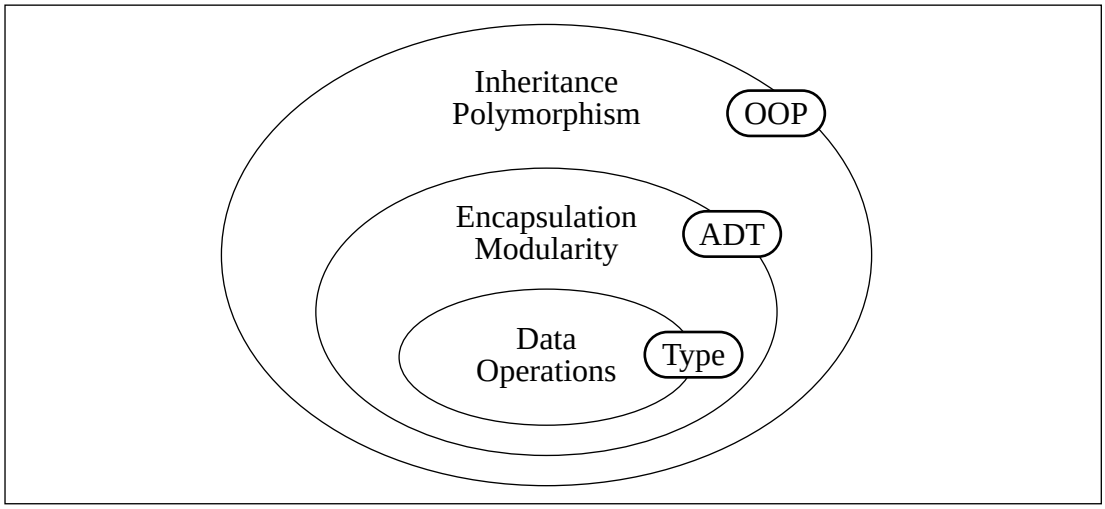

Figure 2.1: Programming Language Concepts

2.1.5 Objects

An object in programming languages is a collection of operations that share a state. The operations determine the messages or calls to which the object can respond, while the state shared among these operations is hidden from the outside world. It is usually accessible only to the object's operations. Variables representing the internal state of an object are called *instance variables*. The operations of an object are often called *methods*. The collection of methods of an object determines its *interface* and behavior.

The object's behavior is entirely determined by its responses to acceptable messages and is independent of the data representation of its instance variables. The operations of an object share its state so that state changes by one operation are "seen" by subsequently executed operations.

2.1.6 Classes

In object-oriented languages, the behavior of objects is specified by classes, which are like the types of traditional languages, but serve additionally to classify objects into hierarchies through the *inheritance* mechanism. Classes serve as templates from which objects can be created by executing a "make-instance" operation. Instantiating a class results in the creation of an object having the same interface as the class, and having a state represented by instance variables as defined by the class. When a client calls a method of an object, the object looks for the method and its implementation in its class definition.

Instantiating two objects of the same class will yield two objects that have each a different identity and a separate state, but share the operations specified in the class definition.

2.1.7 Inheritance

Inheritance is a mechanism for sharing interfaces, code and behavior. It allows reuse of the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Inheritance can express relations among behaviors such as classification, specialization, generalization, approximation, and evolution. Inheritance classifies classes in much the same way as classes classify objects. The ability to classify classes provides even greater conceptual modelling power. Inheritance can be seen as a second-order classification means for sharing, managing and manipulating behavior that complements the first-order management of objects by classes [Weg90].

Abstract classes, also called *virtual classes*, are incomplete behavior specifications that require subclasses to complete their behavior specification before they can be instantiated. Incomplete behaviors are natural building blocks in constructing composite behavior specifications. *Abstract methods* or *virtual methods* are methods without an implementation. They define interfaces common to all subclasses. Concrete subclasses must provide implementations for all abstract methods.

Single inheritance allows a subclass to derive from one parent class only. The resulting hierarchy forms a tree structure. Multiple inheritance, which supports subclasses that share the behavior of several superclasses, gives rise to more complex structures, such as directed acyclic graphs.

2.1.8 Polymorphism

Polymorphism is the ability of several classes of objects to respond to the same message in a similar way. The client that calls a method of an object does not need to know the specific class of the receiver. The semantics of the message will remain the same across many similar classes.

In object-oriented programming languages, polymorphism is reflected in the ability of variables to dynamically denote objects belonging to different classes of a given hierarchy. When invoking a method by means of such a variable, the run-time must determine which actual implementation of the method must be called according to the actual class of the denoted object.

2.1.9 Interactions

Object-oriented programming is a method of implementation in which programs are organized as cooperating collections of objects. To achieve this collaboration, objects may invoke operations on other objects. The result are so-called *client / server* relationships. The invoking object is called the *client*, the object providing the service and executing the

method is called the *server*. In order to make a call to a server object, a client must hold a reference to the server.

2.1.10 Preconditions, Postconditions and Invariants

As we have seen above, one of the main ideas of object-orientation is information hiding. Objects encapsulate state, and client objects may interact with the object only though a well-defined interface. This interface represents the contract between the clients and the object, imposing a certain structure to the way the state of the object can be accessed and manipulated from the outside. This contract can be strengthened to enhance consistency of an object [Mey88].

A *precondition* expresses the constraints under which a method will function properly. A precondition applies to all invocations of the method, regardless of whether the method has been invoked from within a client object, from within the same object or from within an object belonging to a subclass. The precondition binds clients: it defines the conditions under which a call to the method is legitimate.

A *postcondition* expresses properties of the state of an object resulting from a method's execution. It puts an obligation on the implementor of the method. Provided that the method is called with the precondition satisfied, the implementation of the method must fulfil the postcondition.

Whereas preconditions and postconditions describe properties of individual methods, class *invariants* express global properties of all instances of a class, which must be preserved by all methods, or at least by all interface methods. A class invariant can be interpreted as describing the set of all consistent states of its instances.

Preconditions, postconditions and invariants are usually identified during the software development process, especially during the analysis and design phases. The UML specification defines the *Object Constraint Language* [WK99], OCL for short, that allows a designer to specify constraints of this kind for a particular object or method.

When it comes to implementation, these constraints are in general verified by manually inserting assertions into the program code. However, some programming languages, e.g. the object-oriented programming language Eiffel [Mey92], provide more elaborate support for preconditions, postconditions and class invariants.

2.2 Concurrency

The reasons for encountering concurrency in computing systems are two-fold. In a distributed system, concurrency is caused by the fact that the individual components are active. They evolve independently, and sometimes they communicate with each other in order to

synchronize or to exchange data. Concurrency is inherent to distributed systems and can not be avoided.

Centralized systems can also benefit from concurrency. Typical examples of such systems are simulation applications, industrial surveillance systems, or any other system that must handle sporadic incoming events, such as events generated by user interfaces or network traffic. Most of the time, the addressed problem can also be solved using a purely sequential approach. But since in these systems the nature of the problem to be solved is concurrent, designing and structuring such systems by using concurrent activities is intuitive and simple. Concurrency can also be used in centralized systems to improve performance by exploiting multiprocessor architectures.

To handle concurrency, modern operating systems offer two forms of concurrency support. *Processes* (or *heavyweight* concurrency) are programs that usually execute in separate address spaces on a computer system. They can execute concurrently, and the processing power of the system is assigned to the processes following different scheduling policies and priorities. *Threads* make concurrency possible inside a single process (*lightweight* concurrency). Here again, the processing power available to a process is split up among the threads. Processes and threads may take advantage of multi-processor systems.

In the following sections, the term process is used to designate operating system processes or threads.

2.2.1 Nature of Concurrent Systems

According to [LA90, Hoa75, HR73] concurrent systems can be classified into three categories, namely *independent*, *competing* or *cooperating* systems.

In *independent* systems, the individual processes are completely separated from each other. They do not communicate with each other, neither directly or indirectly. They are even not aware of the fact that there are other components running concurrently in the system.

Competitive concurrency exists when two or more processes are designed separately. They are not aware of each other, but share the same resources. Programmers of such processes would like to live in an artificial world in which they do not have to care about other concurrent activities. They want to access objects as if they had them at their exclusive disposal. This form of concurrency is used for example in databases.

Cooperative concurrency exists when several processes cooperate, i.e. do some job together and are aware of this. They can communicate by resource sharing or explicitly. They have been designed together. They cooperate to achieve their joint goal and use each other's help and results.

Real systems are sometimes hard to classify. Often, at some level, active components cooperate, whereas at some other level, they compete for shared resources.

2.2.2 Concurrency and Object-Oriented Programming

Many researchers view all object-oriented systems as inherently concurrent, since objects themselves are “naturally concurrent” entities. In reality, concurrency adds a new dimension to system structure and design. Concurrent systems are extremely difficult to understand, design, analyze and modify.

In general, the execution of a sequential object-oriented program starts by executing a method of a certain object, which in turn calls methods on other objects, etc. At any given time, only one object is executing one of its methods. In concurrent object-oriented programs, several methods can be active at a given time, and even a given method might be invoked multiple times concurrently. Objects must be aware of this concurrency in order to guarantee state consistency in the presence of simultaneous method invocations.

Based on the relationship between processes and objects, concurrent object-oriented programming languages can be divided into two broad categories: *orthogonal* and *integrated* [NP90, TOM99].

Orthogonal languages support concurrent programming by introducing *processes* as special entities different from objects. Processes are active, whereas objects are passive. Processes perform their work by executing methods of objects.

Integrated languages unify processes and objects by defining objects as active entities, eliminating the need for any special mechanisms to define and create processes. An active object encapsulates data as well as one or more processes. Concurrency is introduced in a program by creating active objects.

Integrated languages can be further divided into two subclasses: *homogeneous* and *inhomogeneous*. Homogeneous languages have only one kind of object, namely the active object. Inhomogeneous languages support both active and passive objects. The major drawback of inhomogeneous languages is that the programmer must decide in advance whether to make an object active or passive. This might lead to redundant definitions of classes of very similar behavior and structure when both active and passive objects of a similar kind are needed. The primary advantage is that implementing passive objects is generally simpler and more efficient than implementing active ones. Also, concurrency can be introduced in a selective and controlled manner, so there is no risk of performance degradation when using a large number of objects. The designers of Ada 83 believed that active objects (tasks) as a language construct are enough, and that passive objects are a special case to be dealt with as an optimization issue. Experience has shown that designers of concurrent applications structure their programs differently when passive objects are available, and therefore passive objects have been introduced in Ada 95. A complete overview of the concurrency features present in Ada 95 are reviewed in detail in chapter 10.

The simplest structure for an active object is one in which a single thread is encapsulated within it. In multithreaded active objects, the number of threads within an object may be fixed statically at compile-time or may be changed dynamically at run-time.

[BGL98] gives an extensive overview of the different levels of integration of objects and concurrency.

Java [GJS96] is an example of a language falling into the integrated, inhomogeneous category. Concurrency in Java is supported via the `Thread` class, an instance of which represents a single-threaded object. Programmers can create active objects in two ways:

- A class can extend the `Thread` class and implement the thread's `Run` method.
- A class can implement the `Runnable` interface, which includes a `Run` method. An object with a `Runnable` interface can be passed as a parameter to one of the `Thread` constructors.

A thread object is started by executing its `Start` method. This results in spawning a new thread of control that executes the object's `Run` method.

The major difficulties associated with concurrent programming arise from process interaction. Communication among processes can either be direct, often also referred to as *message passing*, or indirect, based on shared passive objects.

2.2.3 Direct Communication

An active object can communicate with another active object by directly invoking its interface methods. This interaction is viewed as a client / server model relationship.

Communication between the client and the server object can be *synchronous* or *asynchronous* as shown in figure 2.2.

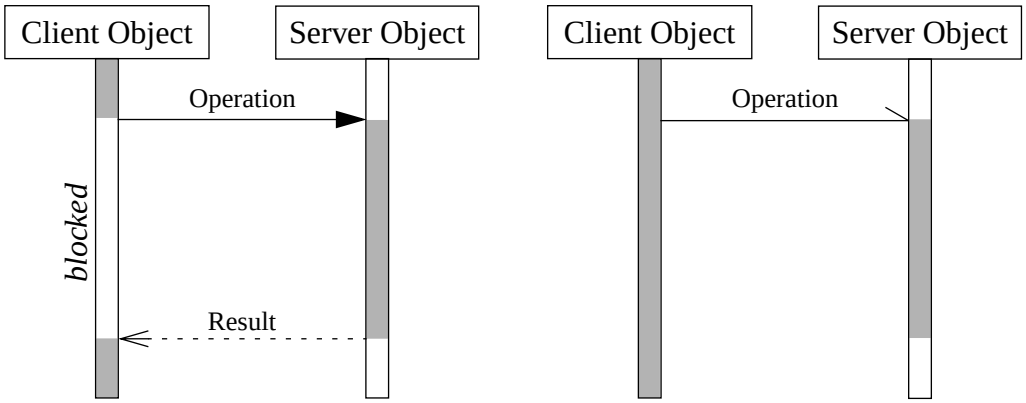

Figure 2.2: Synchronous vs. Asynchronous Communication

When using synchronous communication, the client waits for the method invocation on the server side to complete before continuing its execution. In distributed systems, this model is often used when performing *remote procedure calls* [BN84], RPC for short, or *remote method invocations* in the case of object-orientation. It allows the system to abstract from lower level communication details, e.g. parameter marshalling and message sending, and

hide the inherent concurrency of distributed systems: a remote procedure call, just as a normal procedure call, blocks until the call has been completely executed on the server.

When using asynchronous communication, the client and the server execute concurrently. Returning a result is dissociated from the method invocation.

2.2.4 Communication via Shared Passive Objects

With programming languages that allow communication through shared objects, there are two particularly important classes of synchronization: *mutual exclusion* and *condition synchronization*. The execution of a program implies the use of resources, e.g. files, devices, memory locations, many of which can only be safely used by one process at a time. Mutual exclusion is a synchronization mechanism that ensures that while one process is accessing a resource, no other process can possibly gain access. The sequence of statements that manipulates the resource is called a *critical section*. One means of defining mutual exclusion is to treat a critical section as an indivisible operation. The complete actions on the resource must therefore have been performed before any other process is allowed to execute any, possibly corrupting, action. The *monitor* concept introduced in [BH73] encapsulates a resource and all operations that manipulate the resource. A monitor allows multiple readers or a single writer to access the resource at any given time.

Condition synchronization is necessary when a process wishes to perform an operation that can only sensibly or safely be performed if another process has itself taken some action or is in some defined state. For example, if two processes are communicating via a shared variable, then the receiver of the data must be able to know that the sender has already stored the desired information by executing the appropriate assignment statement. In this case, the receiver wants to synchronize with the sender, but the sender does not need to synchronize with the receiver. If the sender wants to know that the receiver has taken the data, then both processes must synchronize.

In order to provide condition synchronization, C. A. R. Hoare extended the basic monitor concept in [Hoa74] with so–called “condition variables” that could be declared inside a monitor. These basically are signals. A procedure of a monitor may suspend itself by waiting for the condition to become true. Another procedure of the monitor will signal on the condition variable to indicate that the condition has become true. In Hoare’s scheme, the `wait` operation relinquishes exclusion to allow some other process to enter the monitor on the other procedure. The `signal` operation immediately resumes a waiting process if there is one, making the signalling process leave the monitor.

2.2.5 Deadlocks and Starvation

The above synchronizations, although necessary, lead to difficulties that must be considered in the design of concurrent programs. Unfortunately, it is impractical to remove the possibility of these difficulties arising by just providing proper language constructs.

A *deadlock* is a situation in which a set of processes are in a state from which it is impossible for any of them to proceed. This situation arises if the processes form a circular chain, and each process holds resources that are being requested by the next process in the chain.

Different approaches to the deadlock problem can be taken:

- One can attempt to prove that deadlocks are not possible in a particular program under investigation. Although difficult, this is clearly the correct approach to take. Unfortunately, program complexity can make this approach unfeasible.
- Deadlock avoidance algorithms attempt to look ahead and stop the system from moving into a state that will potentially lead to a deadlock.
- Deadlocks can be detected by maintaining wait-for graphs for each resource (see section 7.2.1 on page 80). A cycle in a wait-for graph represents a deadlock. Once a deadlock is detected, the cycle can only be broken by preemptively removing resources from one of the processes.

Livelock or *starvation* is less severe than deadlock. It occurs if a process that wishes to gain access to a resource is never allowed to do so because there are always other processes gaining access before it.

2.3 Fault Tolerance

What exactly is meant by fault tolerance always depends upon the context in which one operates. It is therefore important to first define this context and the domain–specific terms used.

2.3.1 Terminology

To discuss fault tolerance meaningfully, a definition of correct behavior of a program is needed — otherwise, how could one know that something went wrong? For the purposes of fault–tolerant computing, the *specification* of the program is considered to be the definition of correct program behavior: as long as the program meets its specification, it is considered correct.

A *failure* is the observation of an erroneous system state: an observable deviation from the specification is considered a failure. An *error* is that part of the system state that leads to a failure of the system. An error itself is caused by some defect in the system; those defects that cause observable errors are called *faults*. There may be defects in the system that remain undetected; only those that manifest themselves as errors are considered faults. Likewise, an error does not necessarily lead to a failure: it may be a *latent* error [Lap85].

Only when the error in the system state causes the system to behave in a way that is contradictory to its specification, a failure occurs. This relationship is illustrated in figure 2.3.

The goal of fault tolerance is to avoid system failure in the presence of faults. When an error occurs, it must be corrected to avoid a later potential failure: corrective actions have to be taken to restore a correct system state.

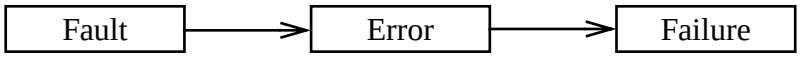

Figure 2.3: Fault Tolerance Terminology

2.3.2 Fault Classification

Faults can be characterized in various ways. One can consider the temporal characteristics of a fault. A *transient* fault has a limited duration, e.g. a temporary malfunction of the system, or a fault due to external interference. If a transient fault occurs repeatedly, it is called an *intermittent* fault. In contrast, *permanent* faults persist, i.e. the faulty component of the system will not work correctly again unless it is replaced.

Another way to classify faults is to consider the software lifecycle phase in which they occur. Here, one can distinguish *design faults* (in particular software design faults) from *operational faults* occurring during the use of the system.

2.3.3 Failure Semantics

Failures, i.e. deviations from a program's specification, can manifest themselves in various ways [Cri91]:

- *Timing* failures can occur in real–time systems if the system fails to respond within the specified time slice. Both early and late responses are considered timing failures; late timing failures are sometimes called *performance* failures.
- *Omission* failures occur when the system doesn't respond to a request when it is expected to do so.
- A *crash* failure occurs when the system stops responding completely. One generally distinguishes *fail–silent* and *fail–stop* behavior: with the latter, the clients of the system have a means to detect that it has failed.
- A system is said to exhibit *byzantine* failure semantics, if upon failure it behaves arbitrarily [LSP82].

These failure semantics can be organized in a hierarchy: byzantine failures are the most general model, and subsume all others as shown in figure 2.4.

The algorithms used for achieving any kind of fault tolerance depend on the computational model, i.e. on what failure semantics we assume for the components in our system.

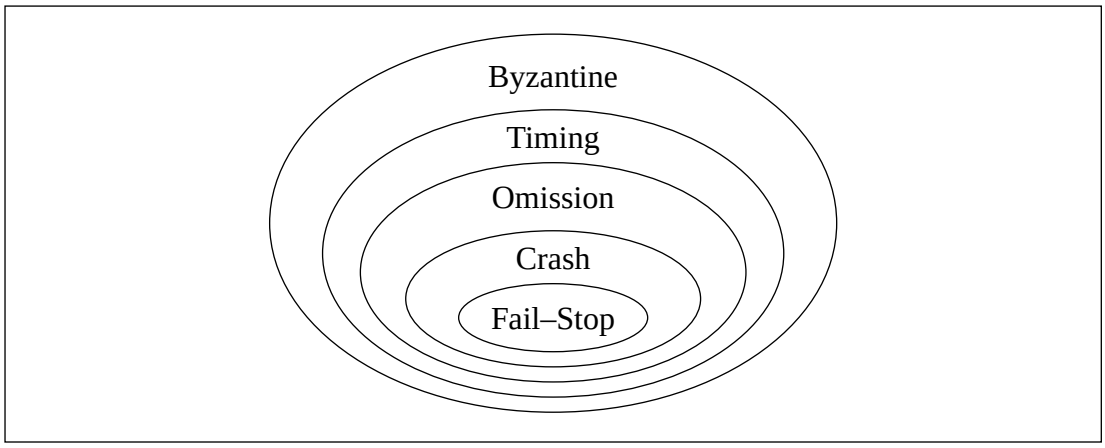

Figure 2.4: Failure Semantics Hierarchy

2.3.4 Error Processing

Once an error has been detected in the system state it should be corrected to avoid a potential system failure later on. Of course, the fault(s) causing the error also should be treated, which means that the reason for the error must be identified and then the defect be corrected in order to avoid that the fault causes more errors. Fault diagnosis and removal is quite different from error processing and is beyond the scope of this thesis.

Once an error is detected, there are several techniques that can be used to deal with it. They can be split into preventive (error *compensation*) and corrective (error *recovery*) measures.

Error masking is the main preventive fault tolerance technique. It exploits redundancy to detect errors and to mask them; a common example is *triple modular redundancy* (TMR): a fault–tolerant component consists of three replicas, the output of the component is the result of some comparator function of the three replicas' individual outputs. Voting (i.e., taking the majority of replies) is one possible comparator function, but depending on the context and the failure semantics of the replicated component, other functions such as taking the average might be adequate.

Corrective methods try to bring the system back into a correct state once an error has been detected. There are two base cases:

- *Forward error recovery* attempts to construct a coherent, error–free system state by applying corrective actions to the current, erroneous state.
- *Backward error recovery* replaces the erroneous system state with some previous, correct state.

Forward error recovery requires that a more or less accurate damage assessment be made. The error must be identified in order to apply corrective actions to prevent system failure. This diagnosis for forward error recovery depends on the particular system. Exceptions (see section 2.4) are provided in programming languages to signal and at the same time identify the nature of an error. Forward error recovery can be achieved through exception handling.

Backward error recovery requires that a previous correct state exists: such systems periodically store a copy of a coherent state (sometimes called *recovery point*, *check point*, *savepoint* or *recovery line*, depending on the recovery technique), to which they can *roll back* in case of an error. Backward error recovery is a general method: because it re–installs a previous, hopefully correct system state, it does not depend on the nature of the error nor on the application's semantics. Its main drawback is that it incurs an overhead even in failure–free executions because recovery points have to be established from time to time. Transactions are a typical instrument for providing backward error recovery. They are presented in detail in chapter 3.

2.3.5 System Structuring for Fault Tolerance

Software systems and systems in general are not monolithic; they usually consist of several components or subsystems, and fault tolerance approaches must account for that. Different approaches may be applied to different components. The composite nature of systems also means that the classification of fault, error, and failure is not absolute: a given component may perceive the failure of a sub–component as a fault and have its own fault tolerance techniques in place to handle it.

This hierarchic model of a system gives rise to the notion of error confinement: the system is structured in regions beyond which the effects of a fault should not propagate undetected. This implies that a given component be accessible to other components only through a well–defined (and preferably narrow [Kop97]) interface. Different error confinement regions may employ different means to achieve fault tolerance. The chosen technique depends upon the failure semantics the system component should adhere to according to its specification, as well as on the failure semantics of its sub–components.

The *Idealized Fault-Tolerant Component* [LA90, RX95] is shown in figure 2.5. The component offers services that may return replies to the component that made a service request. If a request is malformed, the component signals this by raising an interface exception, otherwise it executes the request and produces a reply. If an exception signaling an error occurs, error processing is activated in an attempt to handle the error. If it can be dealt with, normal processing in the component resumes; if not, the component itself signals its failure by an exception. It is immaterial whether exceptions are true exceptions in the sense of exceptions provided by programming languages or are indicated using exceptional replies to requests. It is even possible that some entity external to the system component observes its failure and initiates appropriate error processing in the users of the component.

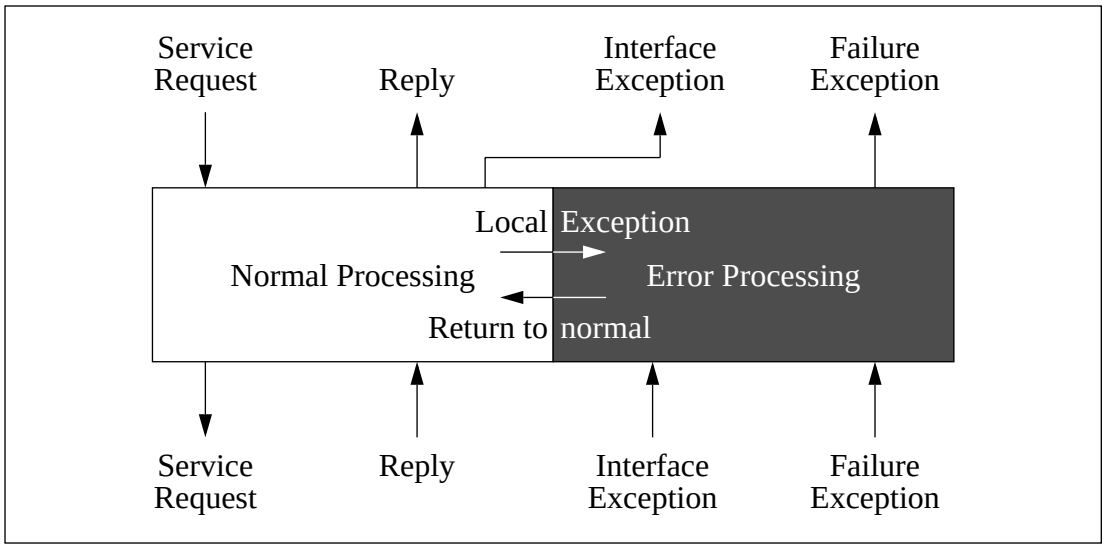

Figure 2.5: Idealized Fault-Tolerant Component

2.4 Exceptions

In order to support fault-tolerance, most modern programming language provide forward error-recovery features by means of exceptions [Goo75]. Exceptions are abnormal events which can happen during program execution. They represent situations in which the normal execution of an operation can not be completed due to some fault [Cri95], e.g. environmental faults, software defects, faults of the underlying software, etc.

Many languages and systems provide special features for handling exceptions in a disciplined way. They allow application programmers to *declare* exceptions and provide them with the ability to treat a program block as an *exception context*. Handlers are associated with such a context, so that when an exception is raised in this context the execution stops and the handler corresponding to the exception is searched among the handlers. Note that there are some models where an exception can be propagated straight to the outside of a context.

Structuring of exception handling is improved if it is possible to differentiate between *internal exceptions* to be handled inside the context, and *external exceptions*, also called *interface exceptions*, which are propagated outside the context. These two kinds of exceptions are not clearly separated in most programming languages although they are intended for very different purposes.

To achieve this separation, the following programming language features are needed:

- Exception contexts are associated with program units;
- Program units have interfaces, where exceptions can be declared;
- Exception contexts can be nested.

The majority of existing exception handling mechanisms use dynamic exception context nesting. In this case the execution of the context can be completed either successfully or by interface exception propagation. In the latter case, the propagated exception is treated as an internal exception raised in the *containing context*. The simplest example of this approach are nested procedure calls. Actually this is the dominating approach suitable for the client / server and the remote procedure call paradigms. It is used in the majority of systems and languages (e.g. in C++ [Str91], Ada [ISO95], Java [GJS96], CLU [LAB+81]). The BETA [MMPN93] programming language on the other hand relies on static exception handling [Knu87].

External exceptions allow programmers to pass in a disciplined, unified and structured fashion different operation outcomes to the containing context. They can be used to inform the higher level context of the reasons for abnormal behavior, and of the state in which the lower level context has been left. The latter is very important, for it makes error recovery at the higher level context possible.

Some systems provide an automatic support which guarantees “all-or-nothing” semantics: if an exception is propagated outside a context, all modifications made inside the context are cancelled. This sort of behavior can be achieved when integrating transactions and exceptions (see chapter 3 for more details). The disadvantage of such systems is of course that such features require complicated run-time support that often results in significant execution overhead. This is why many modern programming languages rely entirely on the application programmer for leaving the context in a known and consistent state.

There is no significant difference between exception handing in object-oriented programming languages and exception handling in block-based programming languages. Exception handling in object-oriented programming languages is in general dynamic, based on nested method calls, or sometimes static, based on objects or classes. The exception contexts are methods or classes, the interface exceptions are declared in the class, often even in the methods’ signatures.

Exception handling integrates very well with object-oriented programming. In some languages, exceptions themselves are objects. They can therefore carry data through attributes, and be extended through inheritance. Object-oriented exceptions make it possible to define exception handlers that handle hierarchies of exceptions.

2.4.1 Exception Handling in Concurrent Systems

Exception handling and concurrency are in general not well integrated in programming languages. Most concurrent object-oriented systems, e.g. Java [GJS96], or Arche [Iss93], provide only classic sequential exception handling. However, concurrency adds a new dimension to system design and execution, and it seems clear that exception handling must live up to this. Concurrent exception handling should be associated with the way a concurrent system is structured, just as it is done for sequential systems. In order to reduce the

complexity of concurrent systems, exception contexts should encapsulate complex behavior consisting of several operations on several objects.

Some researchers have realized the benefits of integrating exception handling and concurrency. The language Facile [TLP+93], an extension of SML, allows the application programmer to declare the same exception in several processes; when the exception is raised in any one of them, the execution of all processes which declared this exception is interrupted and the corresponding handlers are called. If there is no handler for an exception, the process terminates.

The Ada programming language defines synchronous communication between threads by means of the rendezvous concept (see section 10.3.4 on page 135). An exception raised during a rendezvous is propagated out of the accept body into the context of the caller of the rendezvous *and* into the context of the callee containing the accept body.

A more sophisticated example is the concurrent object-oriented language ABCL/1. It has been extended in [IY91] to include concurrent exception handling. This extension relies on the ABCL/1 computational model, within which method calls are viewed as message transmissions between concurrent objects. On the server side, a method call is initiated by accepting the corresponding message. Any method call can be accompanied by a message reply tag indicating the name of the object which will receive the method's results.

In the extended ABCL/1 model, exceptions are treated as signals that can be transmitted between objects. The exception context is a block of statements or a method body. A new notion of *complaint* is introduced. It is similar to the notion of message reply, but intended for informing another object, the complaint destination, of any abnormal situations occurring during method execution. There are four different kinds of complaints: unaccepted messages, time-outs, system-defined and user-defined complaints. A default complaint destination can also be declared for all methods of an object. This mechanism makes it possible to delegate exception handling to an object that is not necessarily the object that invokes the method.

The language εCSP [BI92] also introduces exception handling into concurrent systems. In this language, if a process cannot continue its normal execution because of an exception, it signals a global exception. Any process which subsequently communicates with this process in the course of its normal execution will get the global exception raised in its context.

2.5 Persistence

> *Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and / or space (i.e. the object's location moves from the address space in which it was created).*
>
> *Grady Booch* [Boo91]

There are many possible schemes for supporting persistence. For a complete survey, the reader should refer to [AM95].

The most sophisticated and desired form of persistence is *orthogonal persistence* [ABC+83]. It is the provision of persistence for *all* data irrespective of their type. In a programming language providing orthogonal persistence, persistent data is created and used in the same way as non-persistent data. Loading and saving of values does not alter their semantics, and the process is transparent to the application program.

Whether or not data should be made persistent is often determined using a technique called *persistence by reachability.* The persistence support designates an object as a *persistent root* and provides applications with a built-in function for locating it. Any object that is "reachable" from the persistent root, for instance by following pointers, is automatically made persistent.

2.5.1 Persistence and Programming Languages

The first language providing orthogonal persistence, *PS-Algol* [ACC81], was conceived in order to add persistence to an existing language with minimal perturbation to its initial semantics and implementation. Others have followed this example. There are, for example, persistent versions of functional programming languages such as *Persistent Poly* and *Poly ML* [Mat87]. There has also been work on adding orthogonal persistence to widely used programming languages. Probably the most interesting project nowadays is *PJava* (see section 12.6 on page 180), a project that aims at providing orthogonal persistence to the *Java* [GJS96] programming language.

Due to the demanding requirements of orthogonal persistence, all these implementations had to slightly modify the programming language and / or modify the run-time system. The paper [OC96] investigates adding orthogonal persistence to the Ada 95 [ISO95] programming language. The authors identified the following problems:

- Orthogonal persistence requires that both data and types can have indefinite lifetimes. If a persistent application is to evolve, structural equivalence and dynamic type checking are necessary when a program binds to an object from the persistent store. When introducing orthogonal persistence, type compatibility within an execution extends to type compatibility across different executions. This may conflict with the typing rules of the programming language.
- Often programming languages allow the use of static variables inside classes or even as standalone global variables. It is possible that a programmer uses such static variables to link objects together, such as for instance a static table that links *key* values to some other data. Now if the *key* values are made persistent, the table should also persist, or else the *key* values are useless. It might be tricky to provide automatic persistence for such static variables without breaking orthogonal persistence.

- Orthogonal persistence also requires that elaborate types such as active objects and subprogram pointers persist. This can raise severe implementation problems.
- A program might evolve and change the definition of types and classes, but still try and work with values saved in previous executions. To make this work, some form of version control must be provided, and additional dynamic checking is required. The problem can be even more complicated when considering inheritance.
- Another important problem when providing orthogonal persistence is storage management. Persistent data that will not be used anymore must be deleted, for storage leaks will result in permanent loss of storage capacity. This basically requires some form of automatic garbage collection, at least for all persistent data.

Finally the authors conclude that adding orthogonal persistence to the Ada 95 language would require major changes, making the new language backwards incompatible. It is interesting to note here that even in the case of the Java language, a modern object-oriented language that already provides automatic garbage collection and a powerful reflection mechanism, the virtual machine executing the Java byte code had to be modified in order to support orthogonal persistence [AJDS96].

As soon as one does not require orthogonal persistence, persistence support for conventional programming languages can be provided in multiple ways. Many languages have been extended or provide standard libraries that allow data to be made persistent for instance by saving it to disk. Avalon (see section 12.2 on page 174) for instance is an extension to C++ that provides persistence and transactions. The authors have extended the C++ language, providing additional keywords such as *stable* used to designate class attributes that are to be made persistent.

Persistence support in object-oriented programming languages must provide a mechanism that allows the state of an object to persist between different executions of an application. It can be quite challenging to find a means for taking the in-memory representation of the object's state and writing it to some storage device. Fortunately, object-oriented programming languages often provide some kind of streaming functionality that allows transforming the state of an object into a flat stream of bytes. Some languages go even further and provide streams that allow a user to write objects into files or other storage devices, e.g. Ada `Stream_IO` [ISO95 A.12.1] or Java [GJS96] `FileOutputStreams`. Unfortunately, the facilities provided by the programming language are not always sufficient, or they lack modularity and extensibility, making the definition of new persistent objects or the addition of new storage devices difficult or even impossible.

Chapter 3:
Transaction Models

Complex systems often need more elaborate concurrency features than the ones offered by concurrent object-oriented programming languages (see “Concurrency” on page 14.). The existing single method approaches do not scale well, since they deal with each single operation separately. There is a need for structuring units that encapsulate complex behavior and embrace groups of objects and method calls. These units should represent dynamic system execution as opposed to the static declaration of objects inside objects. System understanding, verification and modification is facilitated if program execution is recursively structured using such units. Examples of applications which require such structuring units are banking systems and e-commerce systems in general, computer supported cooperative work systems (CSCW systems), complex workflow systems, computer assisted design systems (CAD systems), control of modern production lines and cells, etc.

Another concern which makes it necessary to extend the single-object view of system structuring is provision of fault tolerance: in many situations one can not guarantee that erroneous state is confined inside an object. In that case, the application programmer has to deal with very complex error containment domains consisting of several interconnected objects. An error in a server can for example affect several client objects. In order to continue program execution, it is not sufficient to recover only the server or a client. Correct error recovery must recover the system as a whole.

3.1 Atomic Units of System Structuring

Many researchers rely on the concept of *atomicity* in developing structuring approaches for system design. The execution of atomic units is indivisible. Therefore atomic units provide an elegant way to encapsulate state and behavior. No intermediate results can be seen from the outside. The ability to nest such units is extremely important for dealing with system complexity in a scalable way.[1]

The atomicity of an execution has general importance for all phases of system development. It facilitates reasoning about the system, system understanding, verification and development. For instance [Bes96, KSM98] show that concurrent object-oriented systems are easier to understand and to analyze if their execution is built out of atomic units encapsulating several objects and method calls. Providing fault tolerance of different types is also facilitated as these units confine erroneous information [Rom99].

3.2 Atomic Units and Exception Handling

Systems that are structured using atomic units offer a straightforward choice of exception contexts. Atomic units have clearly defined borders, they can be nested, and no information is allowed to cross the border of an atomic unit during its execution. These properties make them ideal candidates for exception contexts.

Exceptions are used to signal abnormal events. In order to recover, all potentially erroneous information must be dealt with during exception handling. The atomicity property makes this task a lot easier, since it guarantees the containment of all potentially erroneous information which must be considered for recovery.

It is important that the way of integrating atomic units and exceptions is compatible with the way exceptions are used in sequential systems. The most natural way is to allow internal exceptions and corresponding handlers to be associated with an atomic unit. The units can have interfaces enriched by external exceptions which the unit can propagate into the containing exception context, i.e. into the containing structuring unit.

There is evidence indicating that it is very likely that multiple exceptions are raised at the same time in a concurrent and, in particular, in a distributed system [RXR96, XRR00]. These complex situations have to be addressed correctly, and atomic units provide a simple and well-structured way of dealing with them.

An overview of exception handling in systems that are structured using atomic units can be found in [RK01].

1. A unit is nested if it contains a subset of objects and method calls of the containing one.

3.3 Competitive and Cooperative Structuring Units

Two different forms of atomic units have evolved: *transactions* and their derivatives which emphasize competitive concurrency, and *atomic actions* and their derivatives which emphasize cooperative concurrency. The authors of [SMR93] name the former *Object and Action* model and the latter *Process and Conversation* model. They claim that the two models are duals of each other, and provide a mapping from one model to the other. Using this mapping, they show that mechanisms used in one model can have interesting counterparts in the other model.

This chapter presents a survey of transaction and atomic action models and analyzes their suitability for concurrent object-oriented programming languages and their integration with exception handling.

3.4 Competitive World: Transactions and Derivatives

Transactions [GR93] are a classic software structure for managing concurrent accesses to global data and for maintaining data consistency in the presence of failures. The notion of transaction has first been introduced in database systems in order to correctly handle concurrent updates of data and to provide fault tolerance with respect to hardware failures [GR93]. A transaction groups an arbitrary number of operations on data objects (from now on called *transactional objects*) together, making the whole appear indivisible as far as the application is concerned and with respect to other concurrent transactions. By using transactions, updates involving multiple transactional objects can be executed as if they happened in a sequential world.

The transaction scheme relies on three standard operations: *begin*, *commit* and *abort*, which mark the boundaries of a transaction. After beginning a new transaction, all update operations on transactional objects are done on behalf of that transaction. At any time during the execution of the transaction it can *abort*, which means that the state of the system is restored to the state at the beginning of the transaction (also called *roll back*). Once a transaction has completed successfully (is *committed*), the effects become permanent and visible to the outside. This approach focusses on preserving and guaranteeing important properties of the data objects (sometimes called *resources*) accessed during a transaction. These properties are referred to as the ACID properties: *Atomicity, Consistency, Isolation* and *Durability* [GR93].

Atomicity
From the perspective of the caller of a transaction, the execution of the transaction appears to jump from the initial state to the result state, without any observable intermediate state — or, if the transaction can not be completed for some reason, it appears as though it had never

left the initial state. Atomicity is a general, unconditional property of transactions. It holds whether the transaction, the entire application, the operating system, or any other components function normally, function abnormally, or crash. For a transaction to be atomic, it must behave atomically to any outside observer. Under no circumstances may a transaction produce a result or a message that later disappears if the transaction rolls back. Atomicity is a vital property for proper system structuring and providing fault tolerance.

Consistency

A transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the application fulfills all the validity constraints of the application according to the applications specification[1]. Unfortunately, this requirement is very hard or even impossible to verify. The state of an application tends to be very complex, and the number of possible consistency constraints among data items is huge. In order to still guarantee consistency, current transaction systems rely on the application programmer to only commit a transaction if the application state has been updated in a consistent way. A transaction must be written to *preserve consistency.* That is, each transaction expects a consistent state when it starts, and recreates that consistency after making its modifications, provided it runs to completion. Note that the intermediate states produced by a transaction during execution of its individual operations need not necessarily be consistent. The transaction system guarantees only that the execution of a transaction will not *erroneously* corrupt the application state.

Isolation

Multiple transactions may execute concurrently. The isolation property states that transactions that execute concurrently do not affect each other, and that the recovery of any of them is separated from the execution of the others. Therefore concurrent transactions produce the same results as if they had been executed sequentially in some order. This does not mean that transactions cannot share objects. It only implies that all modifications that a transaction has made to transactional objects during its execution can not be based on data computed by a yet-to-be-committed transaction.

Durability

Durability requires that the results of a transaction having completed successfully remain available in the future. The system, once it has acknowledged the execution of a transaction, must be able to reestablish its results after any type of subsequent failure. It also implies that there is no automatic function for revoking a completed transaction. The only way to get rid of what a completed transaction had done is to execute another transaction with a counter-algorithm.

1. This assumes that the specification is correct and complete.

Serializability
The mechanism that guarantees the isolation property during transaction execution is called *concurrency control*. It is based on the *serializability* criteria, which states that the results produced by a concurrent execution of a set of transactions must be equivalent to the results produced by executing the same set of transactions serially, i.e. one after the other, in some arbitrary order.

3.4.1 Flat Transactions

Flat transactions represent the simplest type of transaction. A flat transaction contains an arbitrary number of statements encapsulated between a *begin transaction* statement and an "end of transaction" statement, which can either be a *commit transaction* or an *abort transaction* statement. This kind of transaction is called *flat* because there is only one layer of control by the application programmer. Every statement inside the transaction is at the same level; that is, the transaction will either survive together with all modifications made to transactional objects on behalf of the transaction (commit), or it will be rolled back, which means that all changes made to transactional objects will be undone.

The concept of flat transactions is illustrated in figure 3.1. The figure represents a transaction that performs a transfer of money from the bank account A to the bank account B. Both bank accounts are transactional objects. After starting the transaction, the amount of money to be transferred is first withdrawn from account A, then the amount is deposited on account B. If no problems are encountered, the transaction commits.

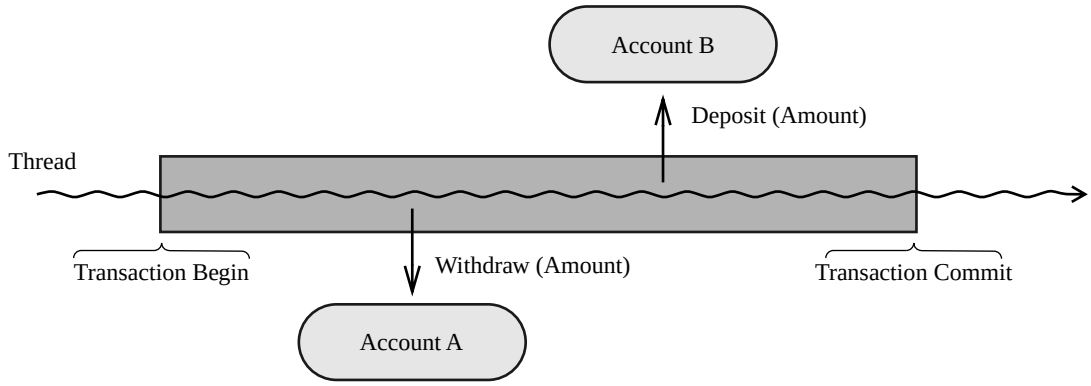

Figure 3.1: A Flat Transaction

Banking systems extensively use transactions, and their importance is nicely illustrated by the transfer example. Without an enclosing transaction, a failure occurring after the withdraw operation has been completed on account A, but before the deposit operation has begun on account B, results in the loss of the amount of money being transferred. Such a situation is not acceptable.

Most systems that only provide flat transactions do not integrate exception handling with transactions, but use return error codes instead. There are many problems with this

approach. Firstly, the use of return codes has always been described as a canonical example of bad practice caused by the absence of a proper exception handling mechanism [Goo75]. Secondly, even if the core language has exception handling, it is not integrated with transactions and, as a result, application exception handling (including the exception context, exception propagation, etc.) is separated from the transactional structure.

3.4.2 Flat Transactions with Savepoints

If some error occurs during the execution of a flat transaction that prevents it from continuing (such as a bank account with insufficient balance, or simply resources that are unavailable), the application programmer has only two choices:

- Perform conventional error recovery, meaning that he or she must manually recover from the error by undoing what went wrong up to a certain point and then re-execute the failed operation or try a different alternate, or
- Abort the transaction as a whole, thereby giving up all changes made on behalf of the transaction so far.

Of course the latter approach is much simpler, and for a short transaction, such as the *Transfer* transaction, it is more appropriate. But there are situations where results are accumulated, not all of which are invalidated by a single error in processing along the way. In that case, giving up all results is undesirable. Having the option of stepping back to an earlier state *inside the same transaction* would be very convenient. This is the idea of *flat transactions with savepoints*.

Inside a transaction, a savepoint can be established by invoking the operation `Save_Work`, which causes the system to record the current state of the transactional objects used so far. The operation returns to the application program a handle that can subsequently be used to refer to that savepoint; it can be passed as a parameter to the operation `Rollback_Work`. As a result, the states of the transactional objects of the savepoint are reestablished. The idea is to establish savepoints at partially consistent states of the application program, which can then be used as restart points when problems are subsequently encountered. The application programmer can then decide to return to the most recent savepoint, or to any other savepoint earlier inside the transaction.

This concept is illustrated in figure 3.2. It shows a flat transaction with three savepoints. After invoking the operation OpA1 on the transactional object A, savepoint 2 is established. Then, operation OpB1 is invoked on transactional object B. At this point, if anything unforeseen happens, the programmer of the transaction has the possibility to rollback to savepoint 2, thus undoing the effects of OpB1, but keeping the effects of OpA1.

Note that the `Begin_Transaction` statement also establishes the first savepoint. There is a difference between aborting a transaction and rolling back to savepoint 1. In the first case, all changes made to transactional objects are undone and the transaction context is

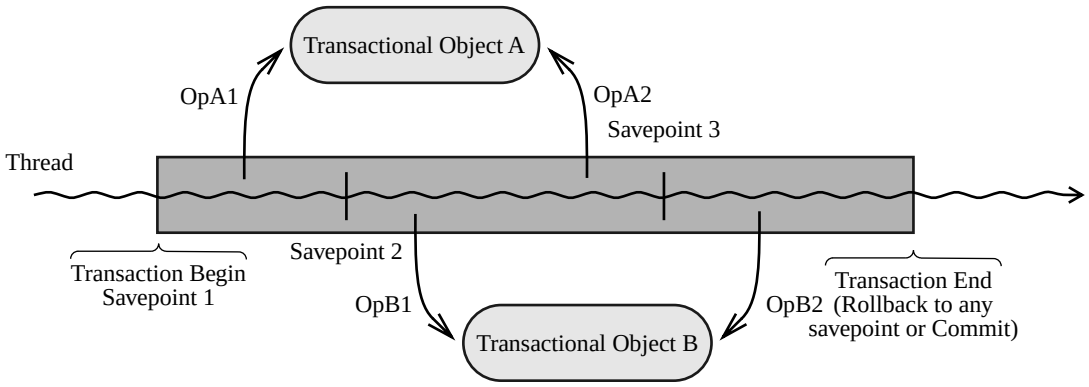

Figure 3.2: A Flat Transaction with Savepoints

deleted. When rolling back to savepoint 1, the transaction stays alive, which means that it keeps the acquired rights on transactional objects and simply returns to the state where it has not yet done anything.

3.4.3 Chained Transactions

Chained transactions are somewhat similar to transactions with savepoints. They try to achieve a compromise between the flexibility of rollback and the amount of work lost after a crash.

The idea of chained transactions is that rather than taking volatile savepoints, the application can commit what it has done so far, thereby giving up the possibility of undoing the changes made to transactional objects. The application, however, can instantly start a new transaction and continue working with the transactional objects without having to reacquire the rights to use them. This request to commit plus the intent to keep going is called `Chain_Transaction`. It is a combination of `Commit_Transaction` and `Begin_Transaction` in one indivisible command. The commitment of the first transaction and the beginning of the next one are wrapped together into one atomic operation. This means in particular that no other transaction can have seen or altered the state of any accessed transactional objects in the mean time. Figure 3.3 illustrates the chained transaction concept.

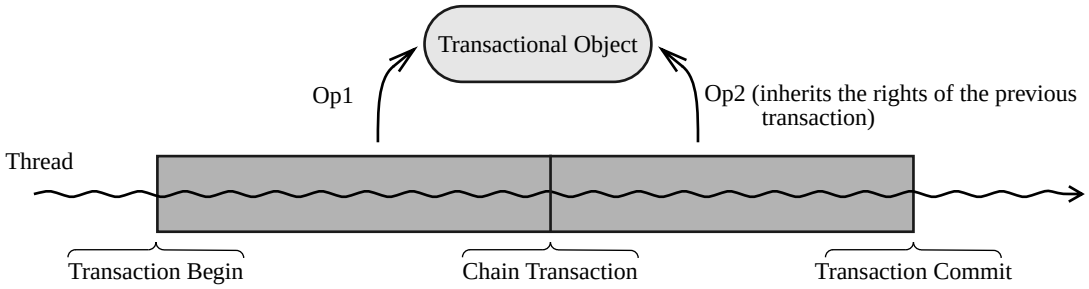

Figure 3.3: Chained Transactions

3.4.4 Nested Transactions

Nested transactions are an extension of the flat transaction model and have first been proposed by Moss [Mos81]. In the nested transaction model, a transaction is allowed to start *subtransactions*, thereby creating a hierarchy of transactions in form of a tree. The transaction at the root of the tree is called the *top-level* transaction. The transactions at the leaf level are flat transactions. A transaction's predecessor in the tree is called a *parent*; a subtransaction at the next lower level is called a *child*. Figure 3.4 shows a parent transaction with three subtransactions.

The rules for nested transactions are summarized below.

Starting Nested Transactions

- At any point in time a new transaction can be created. Creating a new transaction inside some other transaction will start a nested transaction.

Concurrency Control in Nested Transactions

- Accesses to transactional objects from inside a nested transaction are isolated with respect to the parent transaction, to sibling transactions and to other, unrelated transactions.
- All objects held by a parent transaction are made accessible to its subtransactions.

Ending Nested Transactions

- A parent transaction can only commit once all its subtransactions have committed.
- The commit of a subtransaction makes its results accessible only to the parent transaction. Therefore, the changes made to transactional objects are made visible to the outside world only on the commit of the top-level transaction.
- If a transaction aborts, all its subtransactions are also aborted, independently of their local commit status. This rule is applied recursively down the nesting hierarchy. Therefore, if the top-level transaction is aborted, all its subtransactions are also rolled back.

Figure 3.4 shows a top-level transaction T1 with two child transactions T1.1 and T1.2. The second child transaction contains yet another child transaction, transaction T1.2.1. The operation OpB1 is invoked on the transactional object B on behalf of the child transaction T1.1. The effects of the operation are made visible to T1 once T1.1 commits. The transaction T1.2.1 that calls OpB2 later on will therefore operate on the state produced by OpB1. The situation is quite similar for the operations invoked on transactional object A. OpA1 is invoked by the top-level transaction T1. Child transactions are not isolated from their parent transaction, and therefore the child transaction T1.2 is allowed to call OpA2. All changes to A and B are made visible to the outside world once T1 commits.

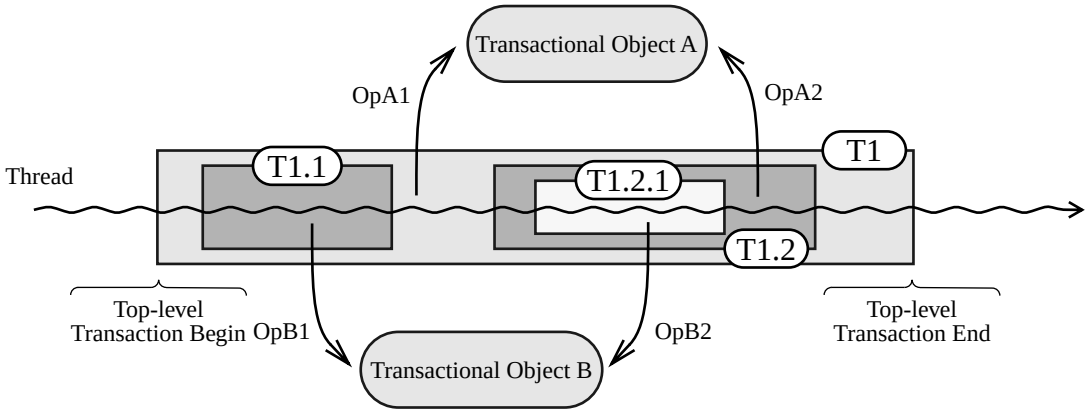

Figure 3.4: Serial Nested Transactions

It is important to note that leaf-level subtransactions are not fully equivalent to classical flat transactions. The key point is that the properties of such transactions are valid only within the confines of the surrounding parent transaction. Leaf-level subtransactions are atomic from the perspective of the parent transaction, they *preserve consistency* with respect to the local function they implement; they are *isolated* from all other activities inside and outside the parent transaction. An immediate consequence of the commit rules is that subtransactions are *not durable*, since their changes are only made persistent when the top-level transaction commits.

Concurrent Execution of Nested Transactions

All transaction models discussed up to this point make use of a single thread to execute operations on transactional objects. The following sections present extended transaction models that increase concurrency by using multiple threads.

In the nested transaction model, sibling transactions can not cooperate, since they are separate transactions, and thus run in isolation from each other. But they can run concurrently, which may increase the performance of the parent transaction dramatically. Most systems implementing nested transactions have realized this. For example Argus (see section 12.1 on page 171) or Camelot / Avalon (see section 12.2 on page 174) provide constructs that allow an application programmer to run sibling transactions in parallel. It is important to note here that the additional threads that are needed to execute the siblings concurrently are created at the transaction boundary. The transactions themselves are still sequential.

Figure 3.5 shows a transaction that performs the transfer operation introduced in the section on flat transactions. This time, the `Withdraw` and `Deposit` operations are performed each in a separate nested transaction. The two sibling transactions are executed concurrently, thus increasing the overall performance. The result of the transfer does not depend on the order in which the two component operations are executed, what counts is that they are either both executed, or none is. This property is guaranteed by the outer transaction. If a

child transaction encounters a problem, it notifies the parent transaction, which in turn can decide to re-execute the failed child transaction, or to abort the entire transaction.

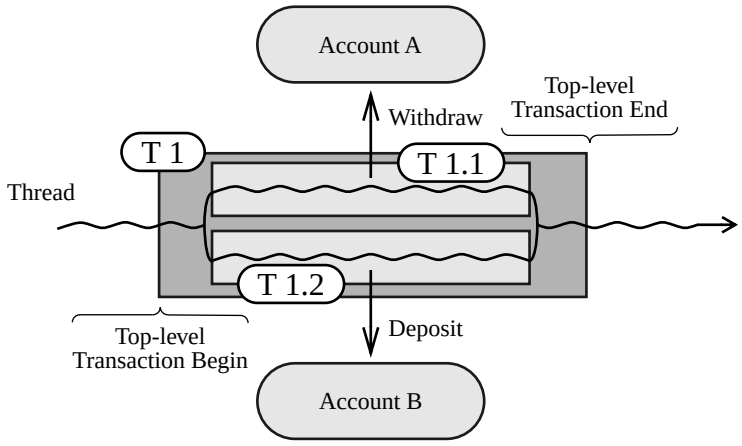

Figure 3.5: Concurrent Nested Transactions

Nested Transactions and Exception Handling

Argus (see section 12.1.3 on page 173) allows a transaction to declare external exceptions in its interface. These interface exceptions are propagated to the initiator of the transaction if the transaction signals it. An exception can be signalled with or without transaction abort, which makes it possible to commit partial results, and to associate different results with different exceptional outcomes.

When executing nested transactions concurrently, exceptions are dealt with separately by each subtransaction. Argus offers also special construct for propagating an external exception out of a group of concurrently executing nested transactions.

3.4.5 Split Transactions

To cope with the problems of long-running transaction as found in CAD/CAM, VLSI design and CASE tools, several additional transaction models have been proposed. They all strive to increase concurrency in order to provide better performance.

The *Split Transaction* [PKH88] model increases concurrency by allowing certain transactions to view the results of other transactions before they commit or abort. Of course, this may create a certain dependency between these transactions as detailed below. In this model, an application programmer may dynamically split a (long) transaction T1, creating a split-off transaction T2 in such a manner that the two resulting transactions are serializable. The application programmer can therefore commit or abort partial results by, for instance, committing the transaction that has been split off (here T2) even before the splitting transaction is committed. In order to make this happen correctly, at the time of the split, operations invoked by T1 up to the split can be divided between T1 and T2, making each responsible for committing and aborting those operations that have been assigned to them. The opera-

tions remaining under the responsibility of T1 may be designated as not conflicting with operations invoked by T2 after the split, and hence T2 can view the effects of these operations. Depending on whether or not such operations have been designated, a split may be *serial* or may be *independent*. In the former case, T1 must commit in order for T2 to commit, whereas in the latter, T1 and T2 can commit or abort independently.

Using this mechanism it is possible to make changes visible to other transactions, even though the transaction that made the changes is still in progress. Splitting also allows other short-duration transactions, which are waiting for objects released as a result of the partial commitment, to proceed.

After the split, T1 can split again. Split transactions can further split, creating a hierarchy of structured transactions different from nested transactions.

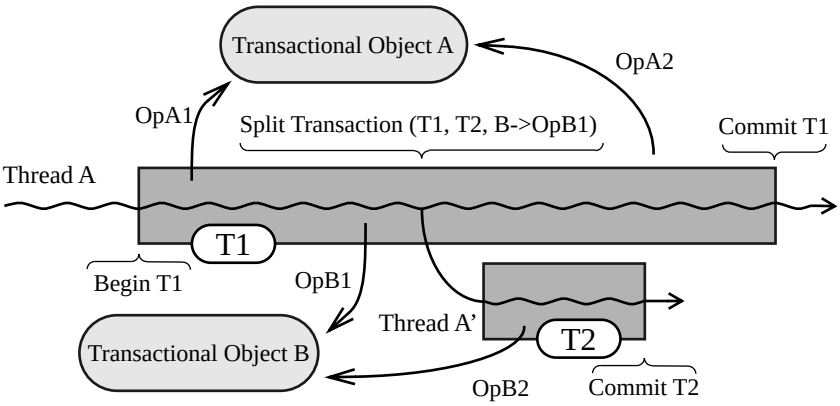

Figure 3.6: Split Transactions

Figure 3.6 shows an example of a transaction T1 and its split-off T2. Before splitting, T1 has invoked operations on the transactional objects A and B. During the split, the responsibility for the operation OpB1 already invoked on B by T1 is handed over to the splitting transaction T2, whereas the right to access the object A remains with T1. Committing T1 will thus result in committing the changes made to A, whereas committing T2 results in committing the changes made to B. After the split, T1 has lost the right to access the external object B.

3.4.6 Joint Transactions

The *Joint Transaction* model [PKH88] is in some way dual to the *Split Transaction* model. In this model a transaction, instead of committing or aborting, is allowed to join another transaction as illustrated in figure 3.7. At the join, the joining transaction T2 releases all its objects, here object B, to the joint transaction T1. The changes made on behalf of T2 are made visible to the outside world only when the joint transaction, here T1, commits; otherwise they are discarded. Thus, if the joint transaction aborts, the operations executed on behalf of the joining transaction are also undone.

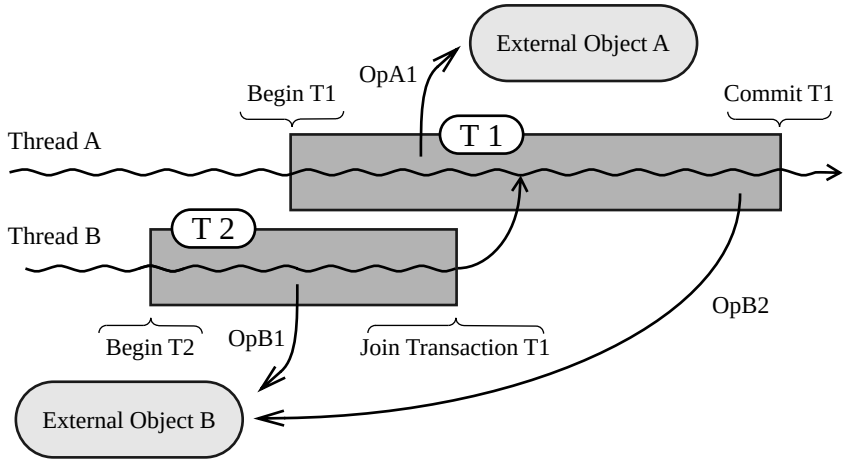

Figure 3.7: Joint Transactions

3.4.7 Recoverable Communicating Actions

The *Recoverable Communication Actions* model proposed in [VRS86] also addresses the issue of long running transactions.

This model allows a transaction (the *sender*) to communicate with another transaction (the *receiver*) by sending results of operations. This communication induces an abort dependency of the receiver on the sender. If the sender aborts, then the receiver must abort as a result of the dependency. Likewise, in order for the receiver transaction to commit, the sender transaction must commit, too.

3.4.8 Sagas

Sagas are another model supporting long-lived activities [GMS87]. A saga is a set of relatively independent *component transactions* T_1, …, T_n, which can interleave in any way with component transactions of other sagas. Component transactions within a saga execute in a predefined order which, in the simplest case, is either sequential or parallel.

Each component transaction is associated with a *compensating transaction*. A compensating transaction undoes, from a semantic point of view, any effects of the component transaction.

Both component and compensating transactions behave like normal transactions, meaning that they have the ACID properties. Component transactions can commit without waiting for any other component transaction or the saga to commit, and hence sagas do not require a commit protocol, as opposed, for example, to nested transactions. Component transactions make their changes to transactional objects visible to the outside world upon commit, and isolation is limited to the component transaction level. Sagas may view partial results of other sagas. Therefore, consistency is not based on serializable executions of sagas.

A saga commits if all of its component transactions commit in the prescribed order. A saga can not execute partially. Thus, when a saga aborts, it has to compensate for the committed component transactions by executing their corresponding compensating transactions. Compensating transactions are executed in the reverse order of commitment of the component transactions.

3.5 Collaborative World: Conversations and Derivatives

3.5.1 Conversations

The concept of a *conversation* has been introduced by [Ran75] in 1975 to achieve software fault tolerance. A fixed number of processes enter a conversation asynchronously; a recovery point is established in each of them. They freely exchange information within the conversation but cannot communicate with any outside process (violations of this rule are called *information smuggling*). When all processes participating in the conversation have come to the end of the conversation, their acceptance tests are to be checked. If all tests have been satisfied, the processes leave the conversation together. Otherwise, they restore their states from the recovery points and may try and execute a different *alternate*. By providing different alternates based on different algorithms that produce the same result allows conversations to tolerate software design faults. This technique is called *software diversity.*

The occurrence of an error in a process inside a conversation requires the rollback of all (and only) the processes in the conversation to the checkpoint established upon entering the conversation. Conversations may be nested freely, meaning that any subset of the processes involved in a conversation at nesting level *i* may enter a conversation at nesting level *i* + 1 [SGR97].

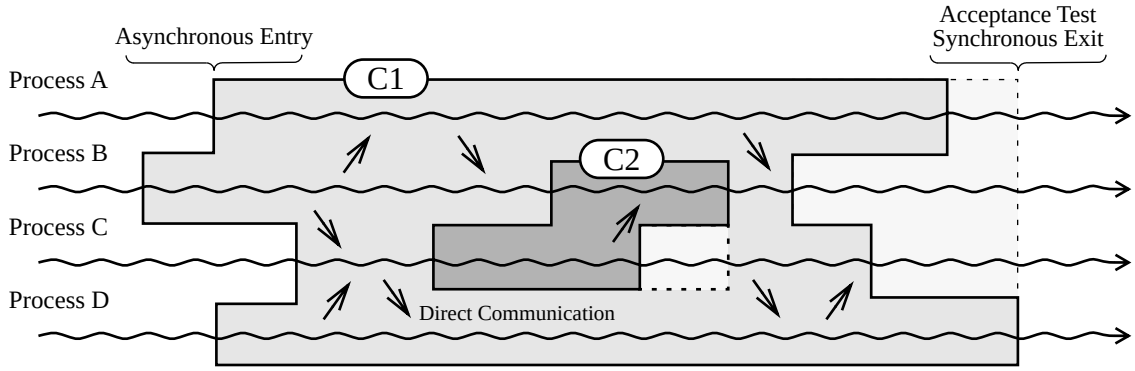

Figure 3.8: Nested Conversations

Figure 3.8 shows four processes entering a conversation C1. Inside, they collaborate by exchanging messages. The processes B and C enter a nested conversation C2. None of the processes is allowed to communicate with the outside world during the execution of the conversation. B and C are not allowed to communicate with A or D during the execution of

C2. Once all processes have reached the end of the conversation, the acceptance test is evaluated. If it fails, all processes must rollback their state to the recovery point established before entering C1.

3.5.2 Atomic Actions

Later on, the conversation scheme has been generalized to tolerate not only design faults, but also hardware faults, transient faults, environmental faults, etc. The resulting model, atomic actions [CR86, LA90], provides additional support for forward error recovery and exception resolution.

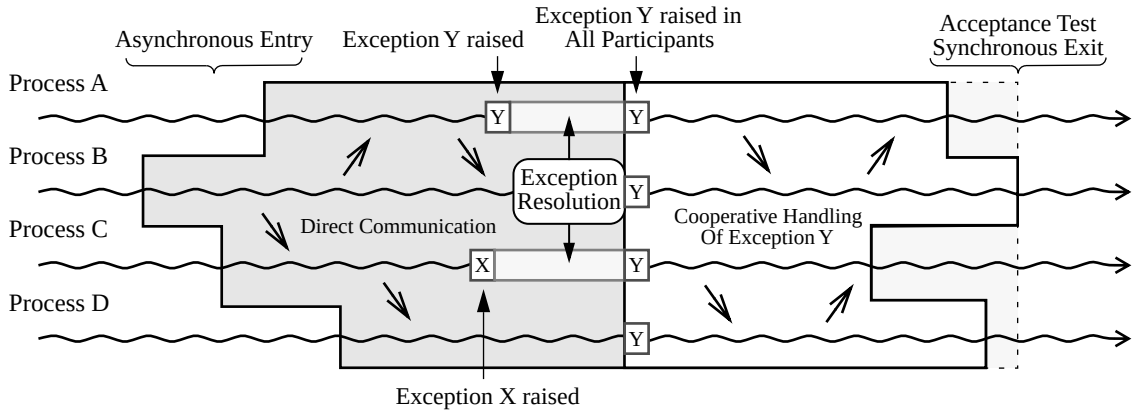

Figure 3.9: An Atomic Action with Coordinated Exception Handling

The structure of an atomic action is represented in figure 3.9. A fixed number of participants (threads, processes or active objects) enter an action and cooperate inside it to achieve joint goals. They are designed to cooperate inside the action and are aware of this cooperation. These participants share work and explicitly exchange information in order to complete the action successfully. Atomic actions structure dynamic system behavior. To guarantee action atomicity, no information is allowed to cross the action border. Actions can be nested, meaning that a subset of the participants of the containing action can enter a nested action. The number of participants of an atomic action is fixed in advance, and hence no dynamic creation of threads is allowed. Participants leave the action together when all of them have completed their job.

Exception Handling in Atomic Actions

A set of internal and external exceptions is associated with each atomic action, and these exceptions are clearly separated. The model is recursive, and all external exceptions of an action are viewed as internal ones of the containing action. Each participant of the action has a set of handlers for all internal exceptions. In this approach, action participants cooperate not only when they execute program functions (i.e. during normal activity) but also when they handle abnormal events. This is mainly due to the fact that when an atomic action is executed, an error can spread to all participants, and the system can be returned into a

consistent state only if all participants are involved in handling. This is why, when an exception is raised in any participant, appropriate handlers are initiated in all of them. An action can be completed either normally (without raising any internal exceptions or after a successful cooperative handling of such exceptions) or by signalling an interface exception to the context of the containing action. Concurrent internal exceptions are resolved using a resolution graph, and handlers for the resulting exception are called in all participants as illustrated in figure 3.9.

The figure represents an atomic action with four participant processes. At some point in time, an instruction executed on behalf of process C raises an internal exception X. Concurrently (or at least before the other participants have been notified of the occurrence of exception X), another exception Y is raised in process A. The exceptions are resolved, and the resulting exception, in our example exception Y, is raised in *all* participants. Corresponding handlers are called in all participants, and after cooperatively handling the exception, the action terminates successfully.

A number of atomic action schemes incorporating different fault tolerance techniques have been developed since then for different languages (such as CSP, Concurrent Pascal, Ada, OCCAM, Java (with and without extensions)), for distributed, multiprocessor and single computer settings, and for different application requirements.

3.6 Combining Cooperative and Competitive Concurrency

3.6.1 Multithreading inside Transactions

With the advent of multiprocessor systems, distributed systems, and programming languages supporting concurrency and distribution, system developers wanted to start using multiple threads of execution inside a single transaction. The concept of using multithreading inside a transaction has been used in different transactional models for quite a long time.

Some systems just allow multiple threads accessing transactional objects on behalf of the same transaction, without paying special attention to this additional form of cooperative concurrency. Figure 3.10 depicts such a transaction. One thread, here Thread C, starts a transaction, then others learn the transaction's identity. By using this identity they can access transactional objects on behalf of this transaction. The model does not restrict the behavior of these threads in any way. They can spawn new threads, or terminate within the transaction. Any thread can commit or abort the transaction at any time (Thread B in figure 3.10). Thread exit from a transaction is not coordinated.

This model is quite general and flexible and has been used in many industrial applications. It leaves thread coordination inside a transaction to the application programmer. Unfortunately this can be dangerous. For example, a thread can decide to leave the transaction and perform some other operations before the outcome of the transaction has been

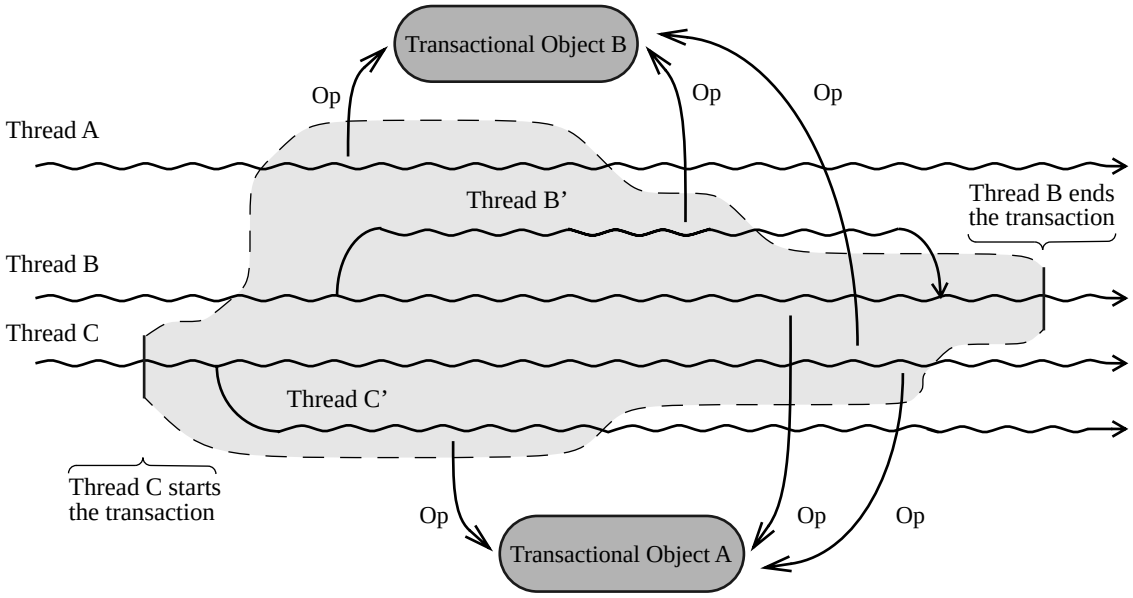

Figure 3.10: Multithreading in Transactions

determined, or a thread can abort the transaction without notifying the other threads. In this model, threads do not actually join the transaction, because the transaction support is not aware of the concurrency. Transactional objects might not be aware of the intra-transaction concurrency either, and therefore consistent execution of concurrent operations is not guaranteed.

Very typical examples using this model are the CORBA Transaction Service (see section 12.8 on page 181) and Arjuna (see section 12.3 on page 176). Generally speaking, a very similar transactional model is provided by the Enterprise Java Beans architecture (see section 12.9 on page 183), EJB for short. Enterprise Java Beans allow system developers to associate several client threads with the same transactional context. Unfortunately, the EJB architecture only supports flat transactions, since nesting is not allowed.

Exception Handling in Transactions with Multithreading

Since there is no well-defined border for these kind of transactions, exception handling is not well integrated into current implementations that use this model. When using the CORBA Transaction Service [Obj00] for instance, a programmer can use only sequential exceptions, i.e. those of the host languages, e.g. C++ or Java. Any exception raised in a transaction can cross the transaction border unnoticed, since the transaction is not the exception context. One cannot define or handle exceptions at the transaction level. Threads working on behalf of the same transaction deal with their exceptions in isolation. Thread coordination in both normal and abnormal situations is the application programmers' responsibility.

It is symptomatic that the designers of Enterprise Java Beans (see section 12.9 on page 183) have made some efforts in combining exception handling and transactions. In this

model, an exception signalled by a transactional object to a thread participating in the transaction can affect the execution of the whole transaction. *System exceptions* will always abort a transaction. *Application exceptions* on the other hand do not automatically abort a transaction, but the programmer can explicitly mark the transaction for abort before raising the exception. Nevertheless, an EJB transaction is not an exception context, and coordination of participating threads, such as notifying them of transaction abort, is left to the application programmer.

3.6.2 Multithreaded Transactions

Multithreaded Transactions first appeared in Venari / ML (see section 12.4 on page 178), and have later also been used in Transactional Drago (see section 12.5 on page 179). In this model, a thread starting a transaction is allowed to spawn additional threads from the inside. Conceptually, the spawning takes places at the transaction border. Before committing or aborting a multithreaded transaction, the forked threads must run to completion. Threads inside a multithreaded transaction can cooperate with each other, but the model does not control or provide any special means of cooperation. However, they can share transactional objects. The transactional objects are aware of this form of cooperative concurrency.

Figure 3.11 shows a multithreaded transaction T1 with a nested multithreaded transaction T1.1.

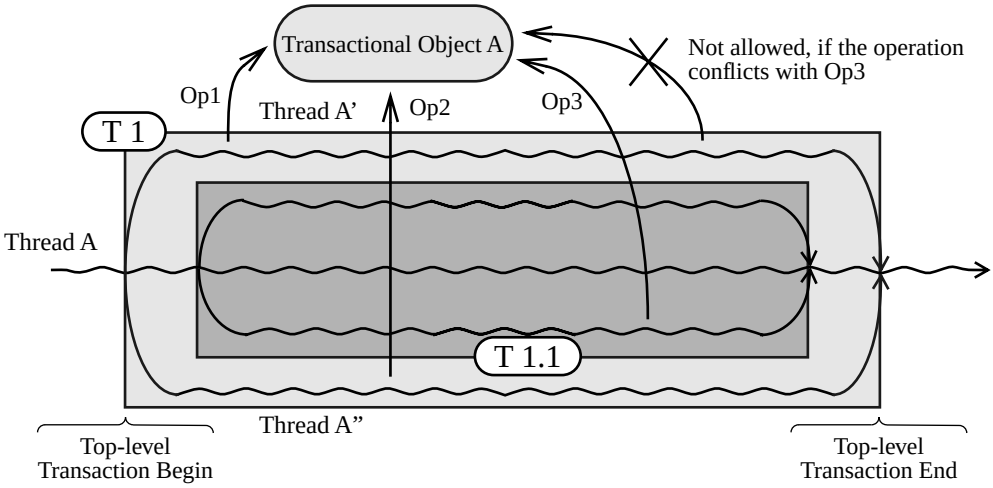

Figure 3.11: Multithreaded Transactions

Thread A starts T1, after which it forks 2 additional threads, Thread A' and Thread A". Thread A' accesses the transactional object A. Later on, Thread A" also accesses the transactional object. This is possible since the threads are both working on behalf of the same transaction. Thread A starts a nested transaction, T1.1, forking again two new threads. One of these threads accesses the transactional object. This is also perfectly legal, since a child transaction may inherit the rights of its parent. There might be a problem later on, if Thread A' tries to access the transactional object again, since child transactions must be run in iso-

lation from their parent. T1.1 commits once all three threads have completed their work. The same rule applies for T1.

Exception Handling in Multithreaded Transactions

Existing systems based on multithreaded transactions incorporate exceptions and transaction in slightly different ways.

The exception handling model of Venari / ML is in many ways similar to that of Argus, but it does not differentiate between external and internal exceptions. It is not possible to declare external exceptions in transactional functions. A transaction is always aborted if any exception is propagated outside of a transactional function. If an exception is raised in a multithreaded transaction without a local thread-level handler addressing the exception, it gets propagated outside the transaction and the transaction is aborted.

Transactional Drago, unlike Argus and Vinari/ML, resolves concurrent exceptions raised by several participating threads before signalling a resulting exception to the outside of the transaction. In this model, external exceptions cannot be declared in the transaction interface, and any exception which is not handled by a thread locally aborts the transaction and gets propagated to the outside.

3.6.3 Coordinated Atomic Actions

The developers of the *Coordinated Atomic Action* (CA action) concept [XRR+95] have defined a model that fully integrates cooperative and competitive concurrency. They have extended the atomic action concept by allowing participants of an atomic action to access external objects. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery, whilst external objects are accessed using transactions in order to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

Each CA action is designed as a stylized multi-entry procedure with roles which are activated by action participants cooperating within the CA action. Logically, the action starts when all roles have been activated and finishes when all of them reach the action end. CA actions can be nested. The state of the CA action is represented by a set of local and external objects. External objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among them. Any sequence of operations on these objects bracketed by the start and completion of the CA action has the ACID properties with respect to other sequences. The execution of a CA action looks like a transaction for the outside world. Action participants explicitly cooperate (interact and coordinate their executions) through local objects.

Figure 3.12 shows a coordinated atomic action with 3 participants.

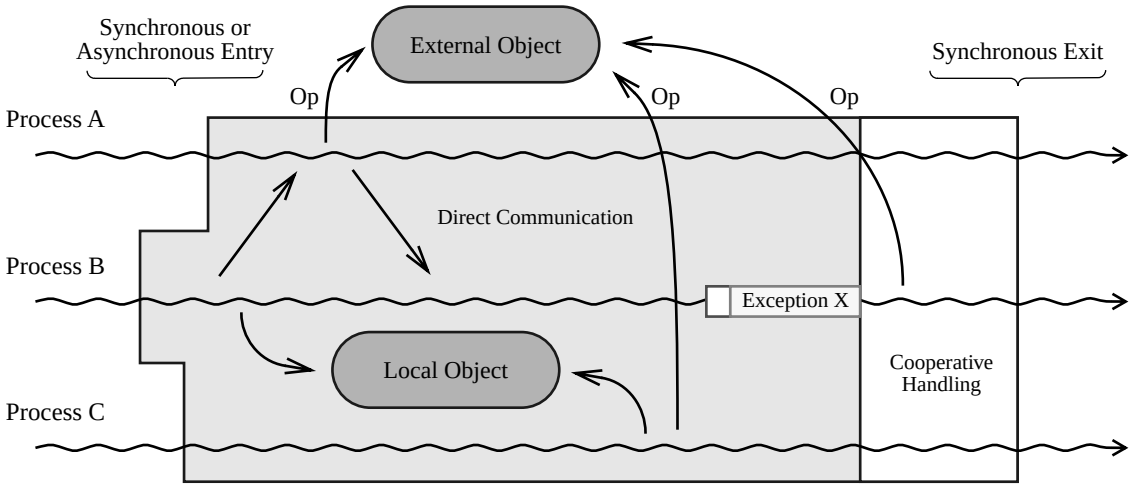

Figure 3.12: A Coordinated Atomic Action

Exception Handling in Coordinated Atomic Actions

All participants are involved in recovery if an error is detected inside a CA action. Conceptually it makes no difference which of them detects the error. The whole CA action represents the recovery region. Exception handling in CA actions is very similar to that in atomic actions: all action participants are involved in cooperative handling of any internal exception, internal exceptions raised concurrently are resolved and external exceptions are explicitly propagated by action participants. Exception handling in CA actions explicitly deals with local and transactional objects [XRR00].

The CA action interface can contain one or more abort exceptions, a predefined failure exception and a number of exceptions corresponding to partial (committed and consistent) results which the action can provide. In the latter case, it uses external exceptions to inform the containing action of the fact that it has not been able to produce a complete required result and, indirectly, of the state in which objects have been left and of the available partial results. If one of the participants signals an abort interface exception, the CA action is aborted; all modifications made to transactional objects are undone and all local objects destroyed. Note that to improve performance, local objects can simply be re-initialized if software diversity or retry are used for recovery. A failure interface exception is signalled by the support when some serious problems are encountered. This might happen if, for example, the support cannot abort or commit the states of transactional objects. When an interface exception corresponding to a partial result is signalled to the outside of an action, the state of all transactional objects is committed before raising this exception in the containing context. In all these cases signalling an interface exception means that the responsibility for dealing with the abnormal event is passed to a higher level in the system structure. At this level, detailed information about the current system state and the reasons for the

exception occurrence can be determined based on the identity of the exception, optional output parameters and the post-conditions associated with the exception.

There has been considerable experimental research on developing object-oriented CA action schemes in Java and Ada and on applying CA actions to developing realistic case studies: a distributed internet *gamma* computation; an *auction system*; a subsystem of a *railway control system* which deals with train control and coordination in the vicinity of a station; and a series of *production cell* case studies, including a fault tolerant one [XRR+99] and a real time one. For the production cell case study, which implements a system controlling a complex industrial application with high reliability and safety requirements, elaborate exception resolution graphs have been built.

Chapter 4:
Open Multithreaded Transactions

This chapter introduces a new transaction model called *Open Multithreaded Transactions* [KRS01] that supports competitive and cooperative concurrency, and integrates well with the concurrency features found in modern programming languages.

4.1 Motivations

A wide range of applications can make use of transactions. As we have seen in the previous chapter, the classic transaction model has been extended in many ways to satisfy the requirements of different application domains.

Using sequential transactions in a concurrent programming language can be very cumbersome. Concurrency must be artificially restricted in accordance with the sequential transaction model. Transactions or subtransactions are allowed to execute concurrently in isolation, but there is no possibility for threads to cooperate and work on behalf of the same transaction.

If concurrency inside a transaction is used together with a transaction support that is not aware of this concurrency, the effects are unpredictable. The state of the transactional objects may get corrupted and information smuggling may occur, compromising the consistency of the application.

The only safe way is to extend the transaction model by integrating concurrency. As a result, the application programmer can make use of concurrency and transactions as he pleases, and does not have to worry about possible interference problems.

Such a model would be of direct benefit to applications that incorporate cooperative and competitive concurrency, dynamic systems that must handle multiple requests in a reliable way, and in general to applications that work in distributed settings and therefore must address fault-tolerance issues. A typical example of such systems are e-commerce applications such as the auction system presented in chapter 13. Other applications that use transactions may also take advantage of such a model by exploiting the possibility of using concurrency inside a transaction to increase performance.

4.2 Requirements

4.2.1 Integration Requirements

When introducing transactions into a concurrent object-oriented programming language, several aspects must be considered. These aspects have been discussed in chapter 2 in detail, and are summarized below:

- *Structure*: In order to make transactions a general and scalable feature for structuring the execution of a system and for providing fault tolerance, nesting of transactions must be possible. This is similar to what is done in object-oriented programming languages, which allow nested method invocations to provide scalable system structuring based on objects.
- *Concurrency*: It is important for the transaction model to support concurrency in a natural way. Inter-transaction concurrency, but also intra-transaction concurrency should be allowed. In particular, the use of the concurrency constructs provided by the language should not be restricted on the inside and outside of transactions, if possible.
- *Exceptions*: Most modern programming languages provide exceptions and sequential exception handling in order to deal with abnormal events. Exception handling is tightly coupled with the structure of a program. Transactions also provide a means of structuring systems, and act as firewalls for errors thanks to the isolation property. Integration of exceptions and transactions is therefore highly desirable.

4.2.2 Guaranteeing the ACID Properties

The classic transaction model guarantees the ACID properties for updates on transactional objects (see section 3.4.1 on page 33). A transaction model for concurrent programming must do so as well.

In general, transactional systems provide atomicity, isolation and durability. The responsibility for consistency is left with the application programmer: a transaction must be written to preserve consistency, moving the application from one consistent state to another one.

Classic transaction models (see section 3.4 on page 31) allow only one thread to work on behalf of a transaction. In order to achieve the desired state change, a thread starts a transaction and sequentially performs operations on transactional objects. If no problems are encountered, the thread commits the transaction and continues execution, once the commit completes successfully. It knows that the transaction committed, and can therefore safely make use of this knowledge on the outside. On the other hand, if anything abnormal happens during a transaction that might compromise consistency, the thread must abort the transaction. The transaction support then performs backward error recovery and undoes all the changes made on behalf of the transaction. The thread then continues execution knowing that the transaction has been aborted.

If multiple threads are allowed to work on behalf of the same transaction, special care must be taken to still guarantee the ACID properties, in particular isolation and consistency. The important issues are summarized below:

- Abnormal situations that might compromise consistency may arise in any of the threads that participate in a transaction, and therefore each of them should be able to abort the transaction.
- In order to prohibit information smuggling, a thread that has successfully completed its work on behalf of a transaction should not be allowed to make use of any information, which has been computed inside the transaction, on the outside, as long as it is not sure if the transaction will commit.
- Transactional objects must be aware of the fact that operations may now be invoked concurrently from within the same transaction. The concurrency control handling isolation between concurrent transactions does not address this issue. To prevent corruption of state, transactional objects must provide additional concurrency control mechanisms.

4.3 Analysis of Existing Models

A model that allows concurrency inside a transaction is the one used by the CORBA Object Transaction Service, Arjuna and Enterprise Java Beans, presented in section 3.6.1 on page 43. Arguably, this model does not really integrate concurrency and transactions; one might better say that concurrency and transactions coexist. The main drawback of this model is that there is no real transaction border, making it hard to guarantee the ACID properties. A thread may conceptually leave a transaction before its outcome has been determined, which may lead to information smuggling if the transaction gets aborted later on. It is also not clear what should happen if during the execution of a transaction a new thread is created and the transaction is aborted later on. Should this new thread be killed? Likewise, must a thread

that terminates inside a transaction be re-created if the transaction aborts? Finally, exceptions and transactions are not integrated in this model.

The multithreaded transaction model (see section 3.6.2 on page 45) comes closest to what is needed to satisfy the requirements discussed in the previous sections of this chapter. One drawback however is that the only way of having concurrency inside a transaction is to start a transaction in one thread, and then spawn new threads inside the transaction. These spawned threads must run to completion before the transaction can be committed. Creation and deletion of threads can be very time-consuming and therefore programmers try to avoid it whenever possible. Process control and especially real-time systems tend to use a static number of threads, created once and for all during initialization of the system. A transaction model for concurrent object-oriented languages should allow existing threads to join an ongoing transaction. Therefore the multithreaded transaction model cannot be used as it does not allow already existing threads to come together and to perform a job on a set of objects in a transactional manner.

On the other hand, we do not want to forbid spawning new threads inside a transaction. This requirement excludes the use of the coordinated atomic action model (see section 3.6.3 on page 46), since it requires the number of participants to be fixed in advance. The kind of collaboration we are looking for is also different from that in the coordinated atomic action model. Participants of a coordinated atomic action collaborate closely; they rely on each other. This is possible, because they know the identity of the other participants and are assured of their presence; they have been designed together and hence are tightly coupled, communicating explicitly or through shared local resources. The collaboration we want to achieve in transactions is somehow different, less entangling. Communication between threads is done exclusively through transactional objects. Also, the number of participant threads is not fixed in advance, since at the beginning of a transaction it may sometimes not even be foreseeable how many threads will participate. In the online auction system example presented in chapter 13, individual auctions are structured using transactions. There will always be a seller thread, but the number of bidder threads is not known in advance. A bidder may want to be able to join an ongoing auction at any time.

These considerations have led to the definition of a new transaction model named *Open Multithreaded Transactions* [Kie99, KR01], borrowing features of the existing models when possible, and adding new features to achieve a seamless integration with concurrent programming. A table showing the relation between open multithreaded transactions and other transaction models is shown in figure 4.3 on page 63.

The following paragraphs describe the rules for open multithreaded transactions. Some of the rules are followed by justifications, which are highlighted using an italic font.

4.4 Open Multithreaded Transactions

Lightweight and heavyweight concurrency are treated in the same way in the open multithreaded transactions model, meaning that what is called thread here might as well be a process executed on a single machine or in a distributed setting. The model allows threads to be created, to run to completion, or to join an ongoing transaction at any time.

There are only two rules that restrict thread behavior:

- A thread created outside of an open multithreaded transaction is not allowed to terminate inside the transaction.
- A thread created inside an open multithreaded transaction must also terminate inside the transaction.

  These two rules avoid the semantic problems that arise when an existing thread enters a transaction and terminates inside it, respectively when a new thread is created inside a transaction and leaves it. The issue here is what happens to these threads if the transaction aborts. If the all-or-nothing semantics of transactions is not only applied to transactional objects, but also to participants, such threads would have to be re-created respectively killed.

Within an open multithreaded transaction, threads can access a set of transactional objects. Although individual threads evolve independently inside an open multithreaded transaction, they are allowed to collaborate with other threads of the transaction by accessing the same transactional objects. In that case they have to be synchronized with respect to the other participating threads in order to guarantee consistency of the accessed transactional objects.

Threads working on behalf of an open multithreaded transaction are referred to as *participants*. External threads that create or join a transaction are called *joined participant*s; a thread created inside a transaction by some other participant is called a *spawned participant*, e.g. in figure 4.1 on page 55 threads A, B, C and D are joined participants, whereas threads B' and C' are spawned participants of the open multithreaded transaction T1.

4.4.1 Starting an Open Multithreaded Transaction

- Any thread can start a transaction. This thread will be the first *joined participant* of the transaction. A newly created transaction is *open*.
- Transactions can be *nested*. A participant of a transaction that starts a new transaction creates a nested transaction. Sibling transactions created by different participants execute concurrently.
- Optionally, the maximum number of participants that are allowed to join a transaction can be specified at creation time. In that case, the transaction *closes* once the maximum number of joined participants has been reached.

4.4.2 Joining an Open Multithreaded Transaction

- A thread can join a transaction as long as it is open, thus becoming one of its *joined participants*. In order to join, it has to learn (at run-time) or to know (statically) the identity of the transaction it wishes to join.
- A thread can join a top-level transaction if and only if it does not already participate in any other transaction. To join a nested transaction, a thread must be a participant of the parent transaction. A thread can participate in only one sibling transaction at a time.
  This rule is necessary for guaranteeing the isolation property. Otherwise, information local to a thread can pass between transactions that should be isolated.
- A thread spawned by a participant will automatically become a *spawned participant* of the innermost transaction in which the spawning thread participates. A spawned participant can join a nested transaction, in which case it becomes a joined participant of the nested transaction.
- A participant of a transaction can decide to *close* the transaction at any time. Once the transaction is closed, no new threads can join the transaction anymore. Note that a participant can still spawn a new thread. If no participant closes the transaction explicitly, it closes once all participants have *finished* (see below).

4.4.3 Concurrency Control in Open Multithreaded Transactions

- Accesses to transactional objects by participants working on behalf of an open multithreaded transaction are isolated from accesses by other transactions. However, participants are allowed to make the identity of the transaction visible to the outside world. This identity can be used by threads willing to join the transaction.
  This rule is one of the main rules guaranteeing the isolation property.
- Accesses to transactional objects by participants of a child transaction are isolated from accesses by participants of the parent transaction.
  This rule extends the nesting rules to allow concurrency while still guaranteeing the isolation property.
- Inside a given transaction, classic consistency techniques, i.e. mutual exclusion, are used to guarantee consistency of transactional objects when accessed by several participants of the same open multithreaded transaction.
  This rule is necessary for guaranteeing the consistency property.

4.4.4 Ending an Open Multithreaded Transaction

- All participants *finish* their work inside a transaction by voting on the transaction outcome. Possible votes are *commit* and *abort*. Voting abort is done by raising an external exception (see below).

- The transaction commits if and only if all participants vote *commit*. In that case, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If any participant votes *abort*, the transaction aborts. In that case, all changes made to transactional objects on behalf of the transaction are undone.
  This rule is important for guaranteeing the isolation and consistency properties. Only changes approved by all participants can be made visible to the outside. If only one participant detects a potential problem that may compromise consistency, the transaction is aborted.
- Once a spawned participant has given its vote, it *terminates* immediately.
- Joined participants are not allowed to leave a transaction, i.e. they are blocked, until the outcome of the transaction has been determined. This means in particular that all joined participants of a *committing* transaction exit synchronously. At the same time, but only then, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If a transaction is aborted, the joined participants may exit asynchronously, but changes made to transactional objects on behalf of the transaction are undone.
  This rule guarantees the isolation property by prohibiting information smuggling. If participants that voted commit were allowed to leave the transaction before its outcome has been determined, they might make use of uncommitted information on the outside.
- If a participating thread "disappears" from a transaction without voting on its outcome, the transaction is aborted, as this case is treated as an error.
  This rule is necessary for guaranteeing the consistency property.

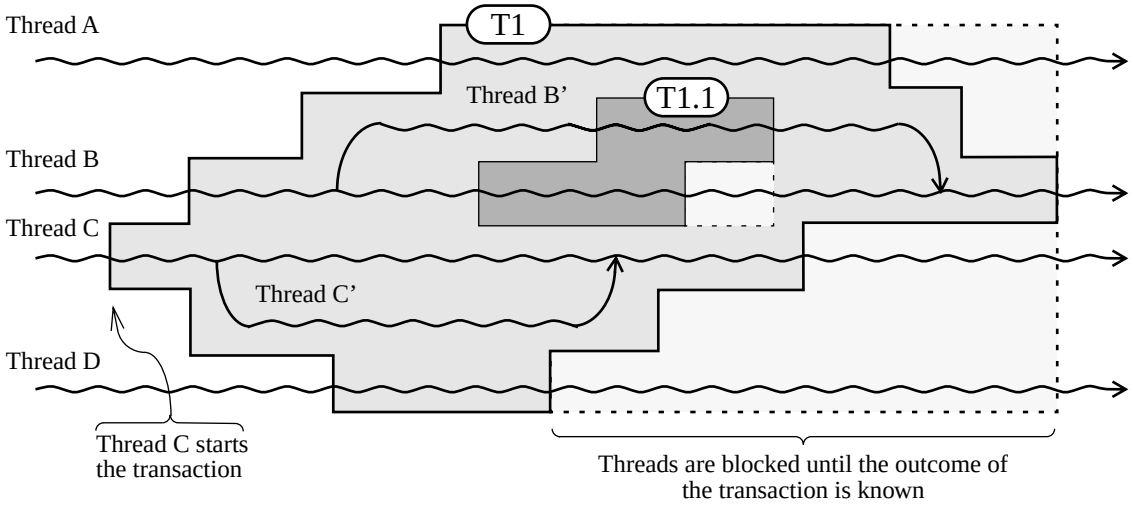

Figure 4.1: An Open Multithreaded Transaction

Figure 4.1 shows two open multithreaded transactions: T1 and T1.1. Thread C creates the transaction T1, threads A, B and D join it. Threads A, B, C and D are therefore *joined par-*

ticipants of the transaction T1. Inside T1 thread C forks a new thread C' (a *spawned participant*), which performs some work inside the transaction and then terminates. Thread B also forks a new thread, thread B'. B and B' perform a *nested transaction* T1.1 inside of T1. B' is a spawned participant of T1, but a joined participant of T1.1. In this example, all participants of T1 vote commit. The joined participants A, C, and D are therefore *blocked* until the last participant, here thread B, has finished its work and given its vote.

4.5 Exception Handing in Open Multithreaded Transactions

This section discusses the exception handling mechanism developed for open multithreaded transactions [Kie00]. Two important design decisions are:

- The model distinguishes *internal* and *external* exceptions; the latter ones are also called interface exceptions;
- Any external exception propagated from a transaction context is interpreted as an abort vote passed by the participant.

The following rules govern exception handling in open multithreaded transactions.

4.5.1 Classification of Exceptions

- Each participant has a set of *internal exceptions* that must be handled inside the transaction, and a set of *external exceptions* which are signalled to the outside of the transaction, when needed. The predefined external exception `Transaction_Abort` is always included in the set of external exceptions.

4.5.2 Internal Exceptions

- Inside a transaction each participant has a set of handlers, one for each internal exception that can occur during its execution.
- The termination model [Goo75] is adhered to: after an internal exception is raised in a participant, the corresponding handler is called to handle it and to complete the participant's activity within the transaction. The handler can signal an external exception if it is not able to deal with the situation.
- If a participant "forgets" to handle an internal exception, the external exception `Transaction_Abort` is signalled.

4.5.3 External Exceptions

- External exceptions are signalled explicitly. Each participant can signal any of its external exceptions.

- Each joined participant of a transaction has a containing exception context.
- When an external exception is signalled by a joined participant, it is propagated to its containing context. If several joined participants signal an external exception, each of them propagates its own exception to its own context.
- If any participant of a transaction signals an external exception, the transaction is aborted, and the exception `Transaction_Abort` is signalled to all joined participants that vote commit.
- Because spawned participants don't outlive the transaction, they cannot signal any external exception except `Transaction_Abort`, which results in aborting the transaction.

Because the open multithreaded transaction model provides transaction nesting, the exception handling rules have to be applied "recursively" by the programmer. All external exceptions of a joined participant are internal exceptions of the calling environment.

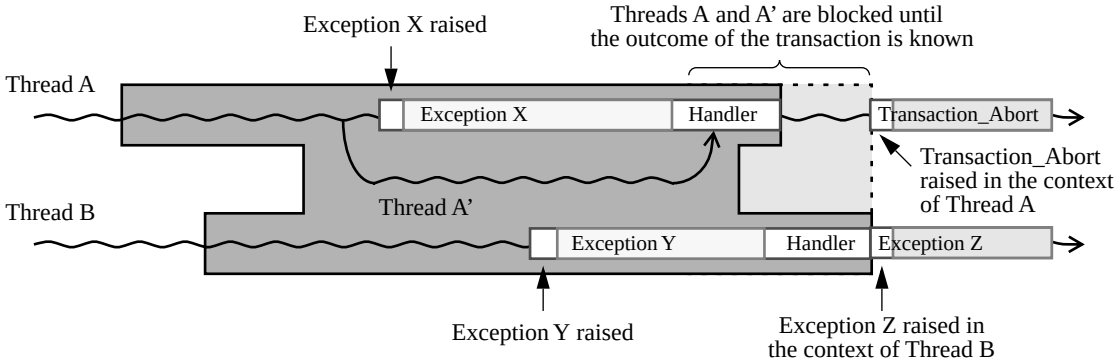

Figure 4.2: Exceptions in Open Multithreaded Transactions

Figure 4.2 illustrates exception handling in open multithreaded transactions. It depicts an open multithreaded transaction with three participant threads. Thread A starts the transaction, thread B joins it, and at some point thread A spawns thread A'. Thread A' performs some work, votes *commit* and *terminates*. Thread A generates an exception while performing its work, but the exception is handled locally. It therefore does not affect the outcome of the transaction; after successful handling, thread A also votes *commit*. Unfortunately thread B has generated an exception, exception Y. It tries to handle the exception, but realizes that it can not recover from this situation. It therefore raises an external exception, exception Z, which causes the transaction to abort. The exception Z is propagated to the calling environment of thread B; in all other joined participants, here thread A, the exception `Transaction_Abort` is raised to notify them of the transaction abort.
If an interface exception has been raised, all participants should be informed about the abort of the transaction as soon as possible. There are two distinct approaches: *non-preemptive* and *preemptive*.

In the non-preemptive approach, each participant completes the transaction by voting commit or by signalling an interface exception in order to vote abort. If a participant votes abort, the other participants get to know that the transaction has aborted only once they vote commit[1]. Non-preemption can decrease performance in applications with long running transactions. If one of the participants of a long running transaction votes abort just after the transaction has been created, all other participants will continue their now useless work until they reach their commit statement.

When using the pre-emptive approach, the transaction support does not wait for the participants to complete, but interrupts all participants as soon as one of them has signalled an external exception. This preemption often requires special run-time support. Its feasibility depends on the mechanisms provided by the programming language or on the underlying operating system. The model does not suffer from performance decrease for long transactions, but unfortunately introducing preemption mechanisms often results in some constant performance overhead even when these mechanisms are not used, i.e. for transactions that commit. Choosing the appropriate model, non-preemptive or preemptive, depends on the characteristics of the application.

4.6 Additional Considerations

This subsection gives the rational for some of the decisions taken during the definition of the open multithreaded transaction model.

4.6.1 Closing an Open Multithreaded Transaction

The open multithreaded transaction model allows a participant to close the transaction, preventing new threads from joining the transaction. This feature has been introduced for two reasons:

- There might be static systems in which one of the participants (most probably the creating thread) knows how many participants are needed to successfully complete the transaction. In that case, it can specify the number of participants during creation of the transaction. As soon as this number of participants is reached, the transaction support automatically closes the transaction.
- In dynamic systems, i.e. systems where at transaction creation time the number of participants is not known, there is a potential for livelock, even though all participants behave correctly. In order to successfully commit a transaction, all participants must vote commit. Without the possibility of closing the transaction, new participants can arrive at any time. This might lead to the situation where all current participants have

1. Of course, if they also signal an interface exception, they have themselves decided to abort the transaction.

decided to commit, but before they can do so, a new participant arrives. It will take some time for this participant to realize that all the work inside this transaction has already been completed. Once it has, it also commits. But during this time, a new participant might have arrived, and so on. In order to prevent this from happening, the transaction must be closed at some point. For some applications, it makes sense to close the transaction as soon as one of the participants has voted commit, other applications might want to leave the decision to a participant that plays a special role (like the *seller* in the auction system example presented in chapter 13).

4.6.2 Naming an Open Multithreaded Transaction

The general model of open multithreaded transactions is asymmetric. In order to set up an open multithreaded transaction with multiple participants, one thread must start the transaction, and only then the other threads can join it. The thread that starts the transaction plays a special role. After starting the transaction, it must somehow tell the other threads that they can now safely join the transaction. On transaction exit, i.e. when voting commit or abort, all participants are treated alike. No synchronization is required among participants; they just vote on the outcome of the transaction once they have finished their work.

This asymmetry can not be avoided if the application requires transactions to be created dynamically. In a static environment on the other hand, a transaction can be associated with a *name*. In this case, there is no need for differentiating starting and joining of such a named transaction. The first thread that expresses the wish to work on behalf of the transaction will effectively start the transaction, the following threads will simply join it.

4.6.3 Deserters

The model requires all participants of an open multithreaded transaction to vote on its outcome. Deserters, i.e. participant threads that terminate without voting, are treated as an error, and cause the transaction to abort.

4.6.4 Transactional Objects

4.6.4.1 Two-level Concurrency Control

In the open multithreaded transactions model access to transactional objects must be controlled at two levels. Guaranteeing the isolation property of all updates made within a transaction with respect to other transactions running concurrently is the first concern. The second concern is ensuring the mutual exclusion of individual operations performed by participants of the same transaction. Generally speaking, the first level can use existing optimistic or pessimistic concurrency control techniques (see section 7.2 on page 80). With the optimistic techniques, where a transaction abort is used to compensate for a consistency

violation, the abort can be either reported to the containing transaction by signaling the `Transaction_Abort` exception or the same transaction can be re-tried (this requires additional run-time support for restoring the thread states). The second level consistency can be guaranteed simply by using monitors or similar techniques found in modern concurrent languages (for example protected objects in Ada 95 (see section 10.3.5 on page 137) or objects with synchronized methods in Java).

4.6.4.2 Enhanced Error Detection

Early error detection is vitally important for modern applications as it makes error recovery faster and more effective. This can be guaranteed only if special methodologies are used while developing systems. In the open multithreaded transaction model the intention is to make the recovery local to individual transaction participants, whenever possible. There are two ways of achieving this within the open multithreaded transaction model: on the participant or on the object sides.

Firstly, all participants of an open multithreaded transaction have to check all parameters which they pass to transactional objects and receive from them when an operation is completed. This is in line with well-known defensive programming.

The second way of enhancing error detection is to develop *self-checking transactional objects*, that is to introduce invariants while designing them. Methods of transactional objects are to be decorated with pre- and post-conditions. When an invariant, pre- or post-condition is violated by the execution of a method, an exception is propagated to the participant that has invoked the operation. There is a considerable body of research related to designing objects / classes together with developing their pre- and post-conditions and invariants, as well as to developing executable conditions to be checked at run-time. The best known example is B. Meyer's “design by contract” methodology supported by features of Eiffel [Mey97] (see also “Preconditions, Postconditions and Invariants” on page 14). This is how transactional objects have to be designed in order to provide early error detection and in order to localize exception handling within one transaction participant.

4.6.4.3 Exception Handling and Transactional Objects

The error detection techniques mentioned above make error containment stronger and increase the chances that an internal exception can be handled locally by a participant. Of course there will still be situations when they fail. In that case the transaction is aborted and all the changes made to transactional objects on behalf of the transaction are undone and an external exception is propagated to the calling context. If additional error recovery is needed, it must be performed at the higher level context.

Just as the open multithreaded transaction model does not support tight collaboration among participants by providing means for direct communication, there is also no automatic cooperative exception handling. Loose collaboration among participants of an open

multithreaded transaction can take place through transactional objects. Exception handling follows the same pattern. If a transactional object propagates an exception to one of the participants during the execution of an operation, it may be left in an erroneous state. If no corrective actions are taken by the participant, then subsequent operation invocations by other participants are likely to raise an exception as well. This situation leads to a form of loose cooperative exception handling, in which the participants may decide to perform a compensation activity on the transactional object. Sometimes, even multiple transactional objects are involved in recovery, since the error might have spread to other transactional objects accessed from within the transaction.

4.6.5 Exception Resolution

As mentioned above, threads inside an open multithreaded transaction cooperate loosely. Each participant thread has its own local exception context, and must handle its exceptions separately. Unlike [PMJPA98, XRR+95] we have decided against some form of coordinated exception resolution for multiple reasons. Firstly, the number of participants of an open multithreaded transaction is not determined in advance, and hence any form of error handling that depends on the presence of participants other than the one that raised the exception can be error-prone. Secondly, exceptions defined in one participant thread might have no meaning or even be undefined in some other participant. Thirdly, there is no need to impose any synchronization among participants, because participants do not cooperate tightly; they act as independently as possible. Finally, concurrent and potentially distributed exception resolution can be very time-consuming and difficult to implement.

4.6.6 Open Multithreaded Transactions as Firewalls for Errors

Open multithreaded transactions are atomic units of system structuring that are intended to move the system from a consistent state to some other consistent state. They are units of error confinement, and as such provide forward and backward error recovery. Application programmers can use exception handlers to catch internal exceptions and then try to address the exceptional situation.

There are three sources of exceptions inside an open multithreaded transaction:

- An internal exception can be raised explicitly by a participant.
- An external exception raised inside a nested transaction is raised as an internal exception in the parent transaction.
- Self-checking transactional objects accessed by a participant of a transaction can raise an exception to signal a situation that violates the consistency of the state of the transactional object.

All these situations give rise to a possibly inconsistent application state. If a participant does not handle such a situation, the application's correct behavior can not be guaranteed.

This is the rationale for providing a default handler for internal exceptions, that simply raises the external exception `Transaction_Abort` and hence will force a roll-back of the entire transaction: the transaction is aborted, the other participants are notified, and the consistent state of the application is restored.

With this behavior, open multithreaded transactions act as firewalls for errors and hence constitute the units of fault tolerance.

4.7 Comparison

Figure 4.3 compares the open multithreaded transaction model with the three main transaction models that allow multiple threads to work on behalf of the same transaction (see section 3.6 on page 43). More details on the systems that use these models can be found in chapter 12.

The highlighted cells in columns 2 - 4 represent the features that the open multithreaded transaction model borrows from the corresponding models. In the last column, the highlighted cells mark the new features introduced by open multithreaded transactions.

Feature	CORBA Transactions, Arjuna, EJB	Multithreaded Transactions: Venari ML, Trans. Drago	Coordinated Atomic Actions	Open Multithreaded Transactions
Nesting	Yes (EJB: No)	Yes	Yes	Yes
Joining	Dynamic	No	Fixed	Dynamic
Forking	Unrestricted	Transaction Bounds	No	Restricted
Commit	One Participant	Main Thread, Waits for Others	All Participants, Blocking Commit	All Participants, Blocking Commit
Exception Integration	No (EJB: A little)	Exception Resolution, Unhandled Exceptions →Abort	Exception Resolution, Internal and External Exceptions	Local Handling Internal and External Exceptions, Unhandled Exceptions →Abort
Transactional Objects	Not Addressed	Two-Way Concurrency Control	Not Addressed	Two-Way Concurrency Control, Self-Checking

Figure 4.3: Comparison of Transaction Models

Part II

The OPTIMA Framework

Chapter 5:
Overall Design

This chapter presents the design of an object-oriented framework named OPTIMA [KJPRPM01] (OPen Transaction Integration for Multithreaded Applications) that provides support for transactions in general, and in particular for open multithreaded transactions. It is a further development of the *TransLib* framework presented in [JPPMA00].

5.1 General Considerations

An object-oriented framework is a set of cooperating classes that make up a reusable design for a specific class of software, in our case transactional systems. As such it defines the architecture of the transaction support, its partitioning into classes and objects, the key responsibilities thereof, and how the classes and objects collaborate. The OPTIMA framework only relies on basic object-oriented and concurrent programming techniques, and can therefore be implemented in any concurrent object-oriented programming language.

Since transactions are used in different software domains, application requirements can differ from one application to another. This is why it is of paramount importance that the framework can be configured by an application programmer to fit the application needs. A transaction support programmer might also want to extend the framework, e.g. by supplying customized concurrency control schemes or adding support for new storage devices.

Section 5.2 presents how such flexibility and extensibility can be achieved using design patterns [GHJV95]. Section 5.3 presents the global overview of the OPTIMA framework. Details about the different components of the framework are given in the following chapters.

5.2 Design Patterns

Well-structured object-oriented architectures are full of patterns that represent solutions to specific recurring problems. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored.

The authors of [GHJV95] have realized this and written a book that contains a collection of design patterns describing simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have been developed, that have evolved over time and proven to be successful. They reflect the experience that programmers have gained during application development, struggling for greater reuse and flexibility in their software. Reference implementations of the most well-known design patterns exist for various programming languages. Design patterns also have the advantage that people familiar with them need less time to understand applications that use them.

The collection of design patterns is constantly growing, and design patterns accepted by the design patterns community are published in the *Pattern Languages of Program Design* book series [CS95, VCK96, MRB98]. The following paragraphs present the design patterns used in the transaction support framework.

5.2.1 The *Abstract Factory* Design Pattern

The *Abstract Factory* design pattern [GHJV95] belongs to the group of *creational patterns*. As such it helps to make a system independent of how its objects are created, using inheritance to vary the class of the object that is instantiated. It decouples the application code that wants to create an object belonging to a "product" class hierarchy from the code that instantiates and initializes a particular object of the hierarchy. This decoupling increases flexibility and extensibility. It lets a programmer configure a system with objects that vary in structure and functionality, statically or dynamically, and also allows him to add additional functionality to the system by extending the product class hierarchy and the factory class hierarchy.

[GHJV95] defines the intent of the *Abstract Factory* design pattern as follows:

> *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.*

Figure 5.1 shows the structure of the *Abstract Factory* design pattern and the names of the different participants. The participants of the design pattern and the roles they play are summarized below:

- *Abstract_Factory* The *Abstract_Factory* class declares an interface for operations that create abstract product objects.

- *Concrete_Factory* The *Concrete_Factory* implements the operations that create concrete product objects.
- *Abstract_Product* The *Abstract_Product* declares an interface for a type of product object.
- *Concrete_Product* The *Concrete_Product* defines a product object to be created by the corresponding concrete factory. The class must of course implement the *Abstract_Product* interface.
- *Client* After instantiating a concrete product object by means of the factory method, the *Client* object works with the product object by calling the methods defined in the *Abstract_Product* class.

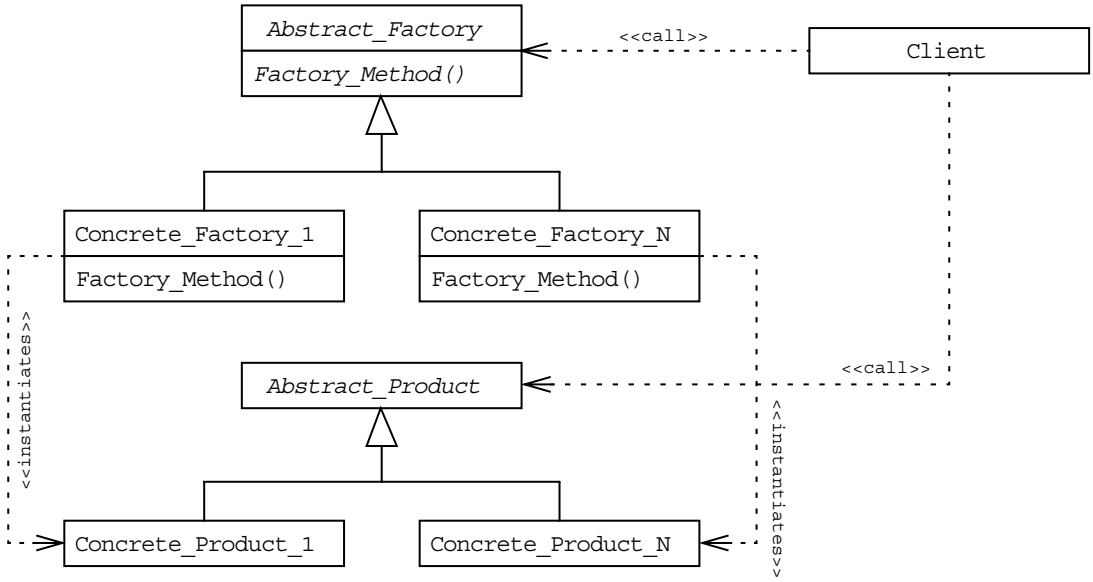

Figure 5.1: The *Abstract Factory* Design Pattern

5.2.2 The *Strategy* Design Pattern

The *Strategy* design pattern belongs to the group of behavioral patterns. Its intent as defined in [GHJV95] is the following:

> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

Figure 5.2 shows the structure of the *Strategy* design pattern and the different names of the participants. The participants of the design pattern and how they interact with each other are summarized below:

- *Strategy* The *Strategy* class declares an interface common to all supported algorithms. *Context* uses this interface to call the algorithm defined by a *Concrete_Strategy.*
- *Concrete_Strategy* The *Concrete_Strategy* implements the algorithm using the *Strategy* interface.
- *Context* The *Context* is configured with a *Concrete_Strategy* object, which it stores as a reference to a *Strategy* object. If necessary, a *Context* class my define an interface that lets *Strategy* access its state.

The *Strategy* pattern is very useful when many related classes differ only in their behavior. By encapsulating the different behaviors in separate classes, it is possible to configure a class with one of these behaviors. Different variants of an algorithm, e.g. reflecting different space / time trade-offs, can be encapsulated and configured at run-time. The hierarchy of strategies can also be easily extended, adding new possible behaviors, without modifying the context classes. The application code in the context class is independent of the concrete strategy implementations, since it only uses the interface defined in *Strategy.*

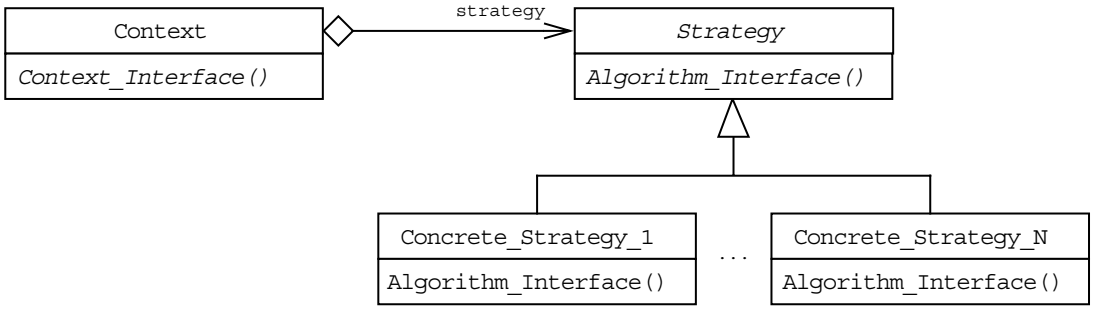

Figure 5.2: The *Strategy* Design Pattern

5.2.3 The *Serializer* Design Pattern

The *Serializer* design pattern described in [RSB+98] is a pattern that is maybe less well known than the previous ones. It provides a mechanism to efficiently stream objects into data structures of any form as well as create objects from such data structures. This is its intent:

> *Read arbitrarily complex object structures from and write them to varying data structure based backends. The Serializer pattern lets you efficiently store and retrieve objects from different backends, such as flat files, relational databases and RPC buffers.*

The structure of the *Serializer* pattern is shown in figure 5.3. The participants of the design pattern and how they collaborate with the other participants are summarized below:

- *Reader / Writer*
  The *Reader* and *Writer* classes declare protocols for reading and writing objects. These protocols consist of read and write operations for every value type, including composite types, array types and object references. The *Reader* and *Writer* hide the *Backend* and the external representation format from the serializable objects.
- *Concrete_Reader / Concrete_Writer*
  The *Concrete_Reader* and *Concrete_Writer* implement the *Reader* and *Writer* protocols for a particular backend and external representation format.
- *Serializable Interface*
  The *Serializable* interface defines operations that accept a *Reader* for reading and a *Writer* for writing. It should also provide a `Create` operation that takes a class identifier as an argument and creates an instance of the denoted class. *Concrete Element* is an object implementing the *Serializable* interface. Such an object can read and write its attributes to a *Concrete_Reader* / *Concrete_Writer.*
- *Backend*
  The *Backend* is a particular backend, and corresponds to our storage class shown in the previous subsection. A *Concrete_Reader* / *Concrete_Writer* reads from/writes to its backend using a backend specific interface. Relational database front-ends, flat files or network buffers are examples of concrete backends.

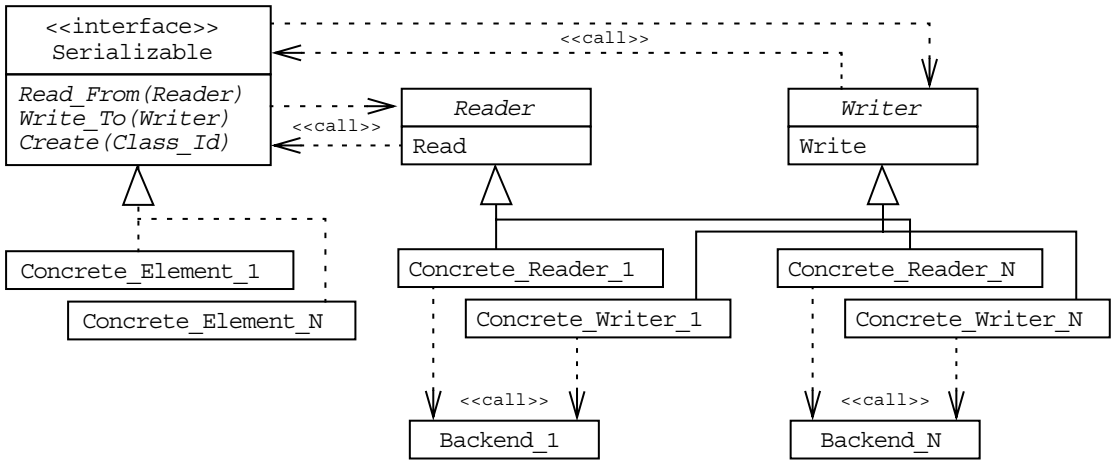

Figure 5.3: The *Serializer* Pattern

When invoked by a client, the *Reader* and *Writer* hand themselves over to the serializable object. The serializable object makes use of its protocol to read / write its attributes by calling the read and write operations provided by the *Reader* and *Writer.* For certain value types such as composite types, the *Reader* and *Writer* might call back to the serializable object or

forward the call to other objects that implement the *Serializable* interface. This results in a recursive back-and-forth interplay between the two parties.

The bigger the set of supported value types of the *Reader* / *Writer* interface is, the more type information can be used by the *Concrete Reader* / *Concrete Writer* to efficiently store the data on the backend. On the other hand, there are backends that support only a small set of value types. Flat files for instance only support byte transfers. For these kinds of backends the *Concrete Reader* / *Concrete Writer* must contain implementation code that maps the `read` / `write` operations of unsupported value types to the ones that are supported.

The big advantage of the *Serializer* pattern is that the application class itself has no knowledge about the external representation format which is used to represent their instances. If this were not the case, introducing a new representation format or changing an old one would require to change almost every class in the system.

In some object-oriented programming languages, such a serialization mechanism is already provided, which means that the `Read_From` / `Write_To` operations defined in the *Serializable* interface have predefined implementations for all value types of the programming language that are not covered by the *Reader* / *Writer* interface. The Java Serialization package [Sun98] and Ada streams (see section 10.7 on page 148) are examples of such predefined language support. If no language support is available, the `Read_From` / `Write_To` operations of the *Serializable* interface must be implemented for every *Concrete Element*.

5.3 OPTIMA Framework Design Overview

A framework providing support for open multithreaded transactions must allow threads running inside transactions to access transactional objects in a consistent manner, guaranteeing the transactional ACID properties: *atomicity, consistency, isolation* and *durability.* At a first glance one might be tempted to design support for these properties separately, but it turns out that most of the properties need a common set of features in order to be implemented correctly. Durability for instance requires some form of persistence support, since the state of a transactional object must be stored on non-volatile storage. After some reflection it becomes apparent that persistence is also needed to guarantee atomicity in the presence of system failures.

The design of the OPTIMA framework can be broken into three important aspects, namely *transaction support, concurrency control* and *recovery support*. They will be introduced briefly in the following subsections, and explained in detail in the following chapters.

How an application programmer will use the framework is also an important issue. Interfaces to the framework, and ways of configuring and extending the framework are presented in chapter 9. The elegance of the interface depends on the features of the programming language. Figure 5.4 gives a general overview of the OPTIMA framework architecture.

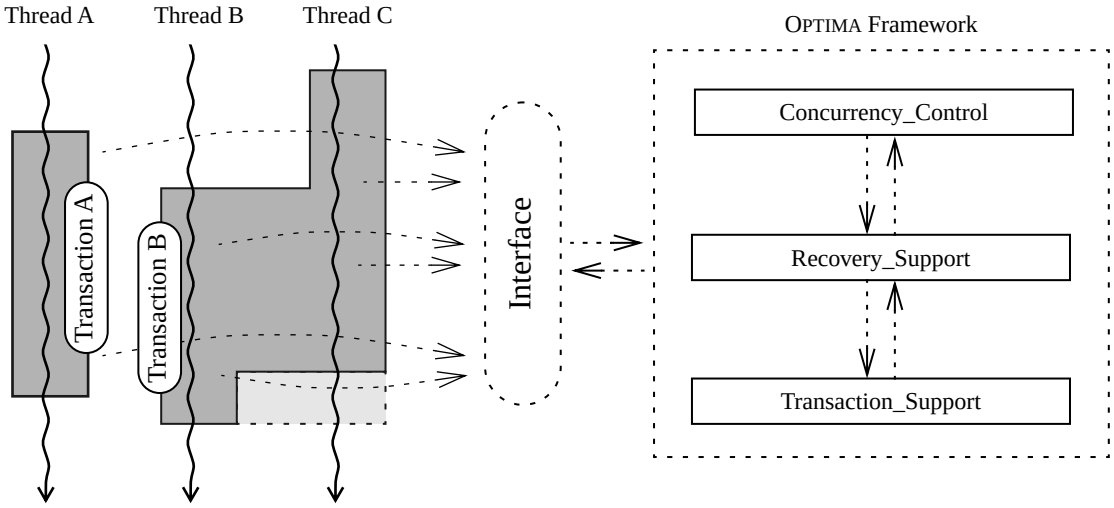

Figure 5.4: OPTIMA Framework Overview

5.3.1 Transaction Support

The transaction support component is responsible for keeping track of the life cycle of an open multithreaded transaction. A newly created open multithreaded transaction starts in the *open* state, is subsequently *closed* (either explicitly or implicitly), and then *aborted* or *committed*. In order to perform these state changes correctly, the transaction support component keeps track of all participants of an open multithreaded transaction by storing their identity and kind, i.e. joined participant or spawned participant. When a participant votes on the outcome of an open multithreaded transaction, the transaction support is notified. If the vote is *commit*, and if there are still other participants working on behalf of the transaction, the calling task is suspended. Only in case of an abort, or if all participants have voted commit, the transaction support passes the decision on to the recovery support.

In addition to participant information, the transaction support also manages the transaction hierarchy, i.e. parent-child relationships among transactions. For each open multithreaded transaction, it must also keep track of all transactional objects that have been accessed by participants.

5.3.2 Concurrency Control

The main objectives of the concurrency control component is to handle cooperative and competitive concurrency in open multithreaded transactions. Dealing with competitive concurrency comes down to guaranteeing the *isolation* property for each transaction. Transactions running concurrently are not allowed to interfere with each other; participants of a transaction access transactional objects as if they were the only threads executing in the sys-

tem. Handling cooperative concurrency means insuring data *consistency* despite concurrent accesses to transactional objects by participants of the same transaction.

These problems can be solved by synchronizing the accesses to transactional objects made by concurrent transactions. Ensuring consistency among participants of the same transaction requires that operations that update the state of a transactional object execute within mutual exclusion. Competitive concurrency control among concurrent transactions can be *pessimistic (conservative)* or *optimistic (aggressive)*, both having advantages and disadvantages. In both cases, the serializability criteria of all transactions must be respected.

The principle underlying pessimistic concurrency control schemes is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. If a transaction invokes an operation that causes a conflict, the transaction is blocked or aborted. This can lead to deadlock situations (see section 2.2.5 on page 18). On the other hand, any transaction that successfully terminates is serializable.

In optimistic concurrency control schemes [KR81], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, transactions are validated to ensure that they preserve serializability. If a transaction is validated, it means that it has not executed operations that conflict with the operations of other transactions, and it then commits.

Concurrency control techniques also differ depending on if only read / write operations are considered, or if more semantic information is available for each operation of a transactional object.

The detailed design of the concurrency control component for open multithreaded transactions is presented in chapter 7.

5.3.3 Recovery

The recovery support provides open multithreaded transactions with *atomicity* and *durability* properties in spite of system failures. Either all modifications made on behalf of an open multithreaded transaction are reflected in the state of the accessed transactional objects, or none is, which means that any partial execution of the modifications has been undone. There are lots of different techniques to perform recovery in case of a system failure, but all save information to a log stored in stable storage [LS79] in order to do so. What information must be stored in the log and when it must be stored depends on when the state of transactional objects are written to their associated non-volatile storage, and again on if only read / write operations are considered or if more semantic information is available for operations of transactional objects.

Chapter 8 addresses the detailed design of the recovery support for open multithreaded transactions.

Chapter 6:
Transaction Support

The transaction support is responsible for keeping track of the life cycle of an open multithreaded transaction, i.e. its current state and the set of transactional objects that have been accessed on its behalf. This information is also called the *transaction context*.

Whenever a thread wants to start a new open multithreaded transaction, join an existing open multithreaded transaction, close a transaction or vote on its outcome, it must contact the transaction support. How this is done depends on the interface to the transaction support and is presented in chapter 9.

6.1 States of an Open Multithreaded Transaction

Figure 6.1 shows a state diagram representing the life cycle of an open multithreaded transaction. A newly created open multithreaded transaction is *open*. As long as it is open, other threads are allowed to join the transaction.

A participant of an open multithreaded transaction can decide to *close* the transaction at any time. Once the transaction is closed, no new threads can join the transaction anymore. In order for an open multithreaded transaction to commit, all participants must have voted commit. If only one participant votes abort, the transaction is aborted.

Threads must explicitly join an open multithreaded transaction in order to be part of it. In particular, only participants of an open multithreaded transaction are allowed to close the transaction or vote on its outcome. Therefore, before effectively changing the state of an open multithreaded transaction, the transaction support must verify that the requesting

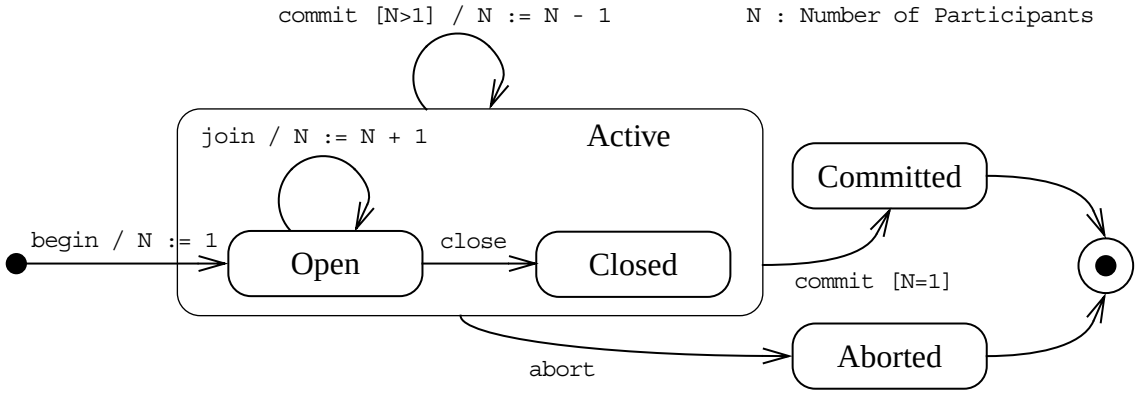

Figure 6.1: Life Cycle of an Open Multithreaded Transaction

thread is a participant of the transaction. It therefore needs to keep track of the identity of each participant.

6.2 Synchronizing Participant Exit

All joined participants of an open multithreaded transaction that commits must exit synchronously. A joined participant that has finished its work on behalf of an open multithreaded transaction by voting commit is not allowed to leave the transaction. This rule prevents results known to participants of an open multithreaded transaction to be revealed to the outside world before the outcome of the transaction has been determined.

In order to implement synchronous exit, the transaction support must have a means to suspend the execution of participants. When a joined participant votes *commit* and there are still other participants working on behalf of the transaction, the participant is suspended. Only in case of an abort, or if all other participants have already voted commit, the transaction support passes the final decision on to the recovery support. If the recovery support is able to successfully commit the transaction, then all suspended participants are released.

6.3 Monitoring Accesses to Transactional Objects

For each open multithreaded transaction, the transaction support must keep a record of all transactional objects that have been accessed by participants of the transaction. This information is needed to perform correct abort or commit processing and to apply recovery in case of a system failure.

6.4 Handling Nesting

In order to support nesting of open multithreaded transactions, subtransactions must know the identity of their parent transaction. This information is needed in two situations:

- When a thread wishes to join an open multithreaded subtransaction, the transaction support must verify that the thread is already a participant of the parent transaction.
- When a subtransaction commits or aborts, the responsibility of all operations made on behalf of the subtransaction must be handed over to the parent transaction.

6.5 The *Transaction* Hierarchy

The functionality described in the previous sections of this chapter has been encapsulated in the `Transaction` class represented in figure 6.2. For each open multithreaded transaction that is started in the system, a corresponding transaction object is created. The object stores the transaction context containing the following data:

- The transaction identity, i.e. a unique identifier, in general a serial number;
- The status of the transaction, i.e. *open*, *closed*, *aborted*, *committed*;
- The current number of participants, their identity and their status, i.e. *joined participant* or *spawned participant*;
- The maximum number of participants of the transaction, provided that such a maximum has been specified;
- A list of subtransactions, and a reference to the parent transaction, if there is one;
- A list of all transactional objects that have been accessed from within the open multithreaded transaction.

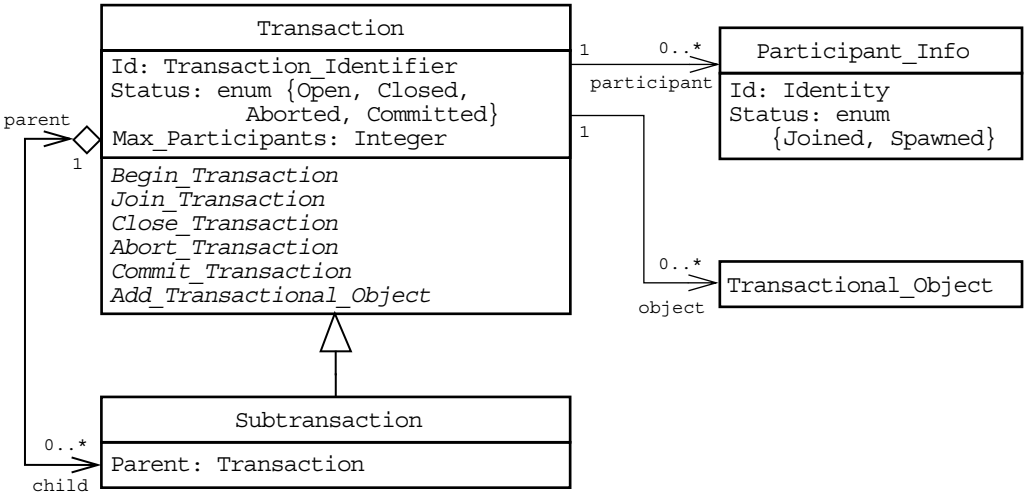

Figure 6.2: The *Transaction* Hierarchy

The `Subtransaction` class inherits from the transaction class, providing an additional association link that designates the parent transaction.

The begin, join, close, abort and commit operations change the state of an open multithreaded transaction according to the state diagram shown in figure 6.1. The `Add_Transactional_Object` method is called each time an operation is invoked on a transactional object on behalf of the transaction (see “Tying Things Together” on page 114).

6.6 Handling Named Transactions

An instance of the `Transaction` class presented in the previous section uniquely identifies a transaction. If transactions are associated with *names* (see “Naming an Open Multithreaded Transaction” on page 59), then the transaction support must provide a way to retrieve the transaction object associated with a particular transaction name.

This functionality is localized in a class named `Transaction_Directory` that provides the operations `Insert_Transaction`, `Delete_Transaction` and `Get_Transaction_Object`.

Chapter 7:
Concurrency Control

The main objective of the concurrency control component of the OPTIMA framework is to handle cooperative and competitive concurrency in open multithreaded transactions.

Participants of an open multithreaded transaction collaborate loosely by accessing the same transactional objects. They are allowed to communicate directly, but this form of communication and synchronization is not supported by the model. Hence, concurrency control in open multithreaded transactions concentrates on the synchronization of accesses to transactional objects by participants:

- Dealing with *cooperative concurrency* means ensuring data *consistency* despite concurrent accesses to transactional objects made by participants of the same transaction.
- Handling *competitive concurrency* comes down to guaranteeing the *isolation* property for each transaction.

Transactions running concurrently are not allowed to interfere with each other; participants of a transaction access transactional objects as if they were the only threads executing in the system. The isolation property guarantees that the abort of a transaction does not cause other transactions to abort. *Cascading aborts* are prevented.

7.1 Handling *Cooperative* Concurrency

Providing consistency among participants of the same transaction requires that operations that update the state of a transactional object (so-called *modifiers*) execute within mutual exclusion. *Observers* — operations that do not change the state of a transactional object —

may execute concurrently. In this respect a transactional object is equivalent to a monitor [Han73].

Cooperative concurrency control should be active for the shortest possible time in order to maximize performance.

7.2 Handling *Competitive* Concurrency

Competitive concurrency control among concurrent transactions can be *pessimistic (conservative)* or *optimistic (aggressive)*, both having advantages and disadvantages. In any case, the serializability of all transactions must be guaranteed.

7.2.1 Pessimistic Concurrency Control

The principle underlying pessimistic concurrency control schemes is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. Typically, a concurrency control manager is associated with each transactional object. Before allowing a transaction to execute an operation on the object, the concurrency manager checks if the transaction performing that particular operation would create a conflict with any other uncommitted operation executed on the object on behalf of other transactions. If a transaction invokes an operation that causes a conflict, the transaction is blocked or aborted. The duration of blocking and the number of times blocking or aborting occurs can be reduced by exploiting operation and object semantics (see “Semantic-Based Concurrency Control” on page 84).

Lock-based protocols use locks to implement permissions to perform operations. When invoking an operation on a transactional object, the caller must first request the lock associated with this operation from the concurrency manager of the transactional object. Before granting the lock, the concurrency manager must verify that this new lock does not conflict with any other lock held by other transactions in progress. If the concurrency manager determines that there indeed would be a conflict, the thread requesting the lock is blocked, waiting for the release of the conflicting lock. Otherwise, the lock is granted, and the thread may proceed and execute the operation.

The order in which locks are granted to transactions imposes an execution ordering on the transactions with respect to their conflicting operations. *Two-phase locking* [EGLT76] ensures serializability by not allowing transactions to acquire any lock after a lock has been released. This implies in practice that a transaction acquires locks during its execution (*1st phase*), and releases them at the end once the outcome of the transaction has been determined (*2nd phase*).

When using blocking pessimistic concurrency control, deadlocks are possible. Two transactions A and B trying to acquire locks on two objects P and Q can deadlock, if A first

requests P and B first requests Q. Now A is waiting for Q, and B is waiting for P. Such deadlocks can be detected and remedied by aborting one of the blocking transactions.

Different deadlock detection mechanisms have been proposed. A common solution is to maintain a wait-for graph for each transactional object. Deadlock detection is initiated either periodically or each time a transaction is blocked by combining the different wait-for graphs of all accessed transactional objects and checking for cycles. A cycle in a wait-for graph represents a deadlock. In such a case, a victim transaction is selected among the blocked transactions. Aborting this transaction frees all the locks held by it, breaking the cycle and allowing some of the blocked transactions to proceed. Another simple deadlock detection strategy is time-out, which aborts a transaction waiting for too long for a lock by simply guessing that the transaction may be involved in a deadlock.

Another solution is to avoid deadlocks. *Timestamp ordering* [BG81] for instance is a pessimistic concurrency control scheme of this kind. The basic idea of timestamp ordering is to associate a timestamp with each transaction, and then process conflicting operations based on the timestamps of the invoking transactions. A transaction is aborted if it attempts to execute an operation after the execution of a conflicting operation invoked by another transaction with a larger timestamp. The concurrency control manager of a transactional object determines whether an operation is in timestamp order by keeping track, for each operation, of the largest timestamp of all transactions that invoked the operation.

Serialization graph testing [Bad79] is another pessimistic concurrency control scheme that avoids deadlocks. In this scheme the concurrency control manager maintains a serialization graph and ensures serializability by aborting the transactions that invoke conflicting operations leading to cycles in the graph. Although serialization graph testing permits any interleaving that is serializable, it requires a computationally expensive solution because it involves cycle testing. In contrast, two-phase locking leads to relatively light implementations, although it disallows the occurrence of certain consistency-preserving interleavings.

7.2.2 Optimistic Concurrency Control

In optimistic concurrency control schemes [KR81], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, the transactions are validated to ensure that they preserve serializability. If a transaction is validated, it means that it has not executed operations that conflict with the operations of other concurrent transactions. It can then commit safely. A distinction can be made between optimistic concurrency control schemes based on *forward validation* or *backward validation*, depending on the manner in which conflicts are determined.

Forward validation checks to ensure that a committing transaction does not conflict with any still active transaction and, consequently, that the committing transaction's effects will not invalidate any active transaction's results. One possibility [HCL90] is to ask all transactional objects involved in a transaction to validate the transaction. Even if only one of

them signals a conflict with some operation executed on behalf of another active transaction, the committing transaction is aborted. This may lead to *wasted aborts* — aborts caused by transactions that get aborted themselves later on.

A different forward validation protocol, avoiding wasted aborts, is *broadcast commit* [MN82]. It guarantees to commit all transactions that reach their commit point. In this strategy, all active transactions that have performed operations conflicting with the validating transaction are aborted. This method avoids wasted aborts, however, instead of aborting only a single transaction at its commit stage as in the previous strategy, many active transactions may be aborted.

Backward validation checks to ensure that a committing transaction has not been invalidated by the recent commit of another transaction. Each object keeps track of Last (*op*), the timestamp of the most recently committed transaction that executed the operation *op*. For each active transaction *t*, each object also keeps track of First (*t*, *op*), the logical time when *t* first executed *op*. An object will validate t if and only if Last (*op'*) < First (*t*, *op*) for each operation *op'* that conflicts with any operation *op* executed by *t*.

In order to avoid rejecting operations that arrive out of order, several concurrency protocols have been proposed that maintain multiple versions of objects [BG81, PK84, AS89]. For each update operation on an object, a new version of the object is produced. Read operations are performed on an appropriate, old version of the object, thereby minimizing the interactions between *read-only* transactions and *update* transactions. Versions are transparent to transactions: objects appear to them as only having a single state.

Many papers have been written describing concurrency control protocols that combine aspects from each of the canonical techniques discussed above (see section 4.5 in [BHG87] for an overview).

7.3 Encapsulating Different Concurrency Control Strategies

Choosing an optimal concurrency control strategy is application dependent. It is therefore important for the framework to allow an application programmer to customize concurrency control to his or her needs. This can be done on a per-object basis using the *Strategy* design pattern (see “The Strategy Design Pattern” on page 69).

The essential concurrency control features are encapsulated in the abstract class `Concurrency_Control` as shown in figure 7.1. It defines the common interface for any concurrency control scheme. At instantiation time, a transactional object can be configured with a concrete concurrency control. The following paragraphs describe the operations provided by the abstract `Concurrency_Control` class.

The `Pre_Operation` and `Post_Operation` operations must be called before respectively after executing any operation on a transactional object. A call to `Pre_Operation` comprises two phases. First, competitive concurrency must be handled. In optimistic concurrency con-

trol schemes based on timestamps for instance, `Pre_Operation` must remember the logical invocation time of the operation. The situation is different in a pessimistic scheme based on locking, where the caller must acquire the lock in order to proceed with the operation. If the lock is not compatible with all other locks granted for this transactional object, the caller will be blocked, waiting for the conflicting transactions to free their resources.

The second phase deals with cooperative concurrency. In both pessimistic and optimistic concurrency control schemes, `Pre_Operation` must acquire the mutual exclusion lock for operations that modify the state of the transactional object. This must be done to guarantee consistency of data modified concurrently by participants of the same open multithreaded transaction. `Post_Operation` releases the mutual exclusion lock again, but does not discard competitive concurrency control information by, for example, discarding the timestamps, or releasing the transaction locks. In general, competitive concurrency control information must be kept until the outcome of the transaction is known.

When the transaction support is ready to commit a transaction, the `Validate` operation is called for each accessed transactional object. In optimistic concurrency control schemes, `Validate` verifies that there are no serializability problems for this transaction by applying forward or backward validation techniques. For pessimistic concurrency control schemes, `Validate` always succeeds.

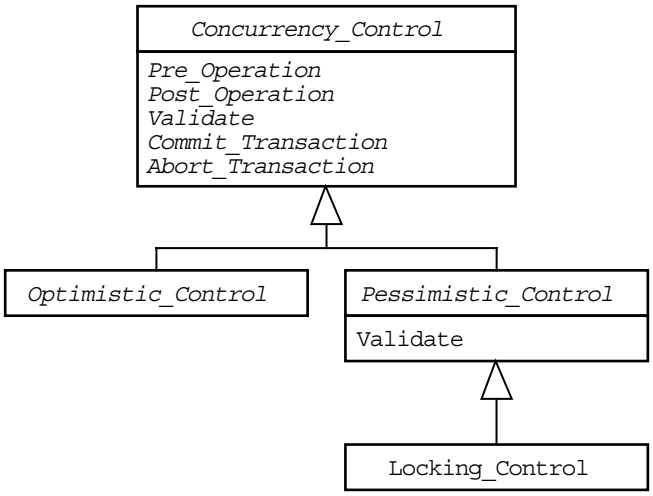

Figure 7.1: The *Concurrency_Control* Hierarchy

7.4 Concurrency Control Information for Operations

In order to correctly handle cooperative concurrency, the concurrency manager must be able to determine for each operation of a transactional object if it is an *observer* or a *modifier*. To deal with competitive concurrency, optimistic and pessimistic concurrency control schemes must be able to decide if there are conflicts between operations invoked by different transac-

tions that would compromise the serializability of these transactions. This information must be associated with each operation of a transactional object.

The following sections introduce strict concurrency control and semantic-based concurrency control for operations, and how they are integrated into the OPTIMA framework.

7.4.1 Strict Concurrency Control

The simplest form of concurrency control among operations of a transactional object is *strict concurrency control*. In locking based concurrency control schemes this technique is also referred to as *read / write locking*.

It is simple, for strict concurrency control only distinguishes observer and modifier operations. Reading a value from a data structure does not modify its contents, writing a value to the data structure does. In this case, cooperative and competitive concurrency control are based on the same criteria.

The compatibility table of read and write operations is shown in figure 7.2.

	`Read (x)`	`Write (x)`
`Read (x)`	yes	no
`Write (x)`	no	no

Figure 7.2: Compatibility Table of *Read* and *Write* Operations

Some programming languages explicitly show the difference of observer and modifier operations in the signature of the operation. Protected types in Ada distinguish functions, which can only read values encapsulated in a protected object, and procedures, which on the other hand are allowed to modify the encapsulated value (see section 10.3.5 on page 137).

Depending on the different constructs that are available to an application programmer and depending on the visibility rules of the programming language, the compiler may be able to automatically determine if a operation modifies data values encapsulated inside an object or not.

7.4.2 Semantic-Based Concurrency Control

Inter-transaction concurrency can be increased if one knows more about the semantics of the operations of a transactional object. Exploiting this knowledge can drastically increase the performance of an application that uses transactions.

According to [RC97], the concurrency semantics of a transactional object depend on the following:

- Semantics of the operations,
- Operation input and output values,

- Organization of the object, and
- Object usage.

Operation semantics are related to the effects of an operation on the state of the transactional object. As mentioned before, operations can be broadly classified as *observers*, which do not change the state of an object, and *modifiers*, which do change the state of the object.

Input / Output semantics refer both to the direction of information flow between a client and a transactional object, and to the meaning of input and output values of an operation. The information flows into and out of an object occur via the arguments of the operations defined on the object and through the return values of these operations.

Object organization semantics refer to the abstract organization of an object. It can be further composed into *composition* semantics, which pertain to what an object is composed of, and *order* semantics, which refer to the relative ordering among the component objects. It is obvious that if an object contains two separate data structures, operations that work only on one of them do not conflict with operations that work on the other one.

Usage semantics refer to how the object is used and what is done with the information extracted from an object by an operation invoked by a transaction. This depends on the structural and behavioral semantics of the transaction, and on the application.

The following section discusses how to increase inter-transaction concurrency by exploiting the first three characteristics of transactional objects. The last characteristic can not be exploited on a per-object base, since the usage of an object depends in general on the application.

7.4.2.1 Commutativity

Lets consider an abstract data type representing a set. A set is a non-ordered collection of elements without duplicates, meaning that for a given element there can only be one instance in the set at a given time. A set provides three operations, `Insert(Set,Element)` to insert an element into the set, `Remove(Set,Element)` to remove an element from a set, and `Is_In(Set,Element)`, an operation that tests if a certain element is part of a given set or not.

`Insert` and `Remove` are modifier operations, `Is_In` is an observer operation. At first sight one might think that two invocations of `Insert` conflict with each other. If we take into account the input / output semantics of the `Insert` operation, we soon realize that two invocations of the insert operation for different elements, `Insert(Set,A)` and `Insert(Set,B)` do not conflict with each other. The resulting set does not depend on the sequence in which the two insertion operations are executed. This property, allowing to interchange two operations and still get the same result is called *commutativity.* Two operations *commute,* if their effects on the state of a transactional object and their return values are the same, irrespective of their execution order.

In order to apply commutativity, every transactional object must provide a compatibility table that precisely states for each operation of the object the conditions under which an

invocation of the operation commutes with the other available operations. When a transaction invokes an operation, the concurrency control can consult this table and determine if there is a conflict by verifying that the operation commutes with every uncommitted operation executed on the object so far. The commutativity table for the set operations is shown in figure 7.3.

	`Insert (y)`	`Remove (y)`	`Is_In (y)`
`Insert (x)`	$x \neq y$	$x \neq y$	$x \neq y$
`Remove (x)`	$x \neq y$	$x \neq y$	$x \neq y$
`Is_In (x)`	$x \neq y$	$x \neq y$	yes

Figure 7.3: Backward Commutativity Table for the *Set* ADT

Depending on the update strategy used for transactional objects (see section 9.2.5 on page 115), two slightly different forms of commutativity must be provided. *Backward commutativity* is used in combination with immediate update of data objects. In this scheme, each operation is immediately executed on the transactional object, possibly modifying its state. The ordering in which two operations A and B are executed on a transactional object is important in this case, since the operation executed second “sees” the results of the execution of the first one. B commutes with A, if A followed by B has the same effects as executing A, then B and then undoing A, irrespective of the initial state of the transactional object. In particular, the return values of B must be the same in both cases.

Forward commutativity is used in combination with deferred update of data objects. In this scheme, each operation on a transactional object is executed on a separate copy of the state of the object. The ordering of the operations A and B is not important in this case, since they both “see” the same state of the object. B commutes with A, if B’s return values do not dependent on the modifications that A applies to the state of the transactional object.

The difference between these two forms of commutativity can be illustrated on the set example. In both schemes, two insert operations that insert *different* elements into the set commute. However, if the inserted elements are *identical*, the two operations commute only if deferred update and hence forward commutativity is used. When using immediate update, the first insert operation `Insert(Set,A)` will insert the element A into the set. The second invocation of `Insert(Set,A)` has no effect on the set, since the set already contains the element A. If the transaction that executed the first insert operation aborts, the insert operation must be undone. Unfortunately, undoing the first operation will *remove* the element A from the set, thereby also undoing the second insert operation.

7.4.3 Encapsulating Operation Concurrency Control Information

The concurrency control component of the OPTIMA framework provides support for strict concurrency control and semantic-based concurrency control.

Optimistic and pessimistic concurrency control schemes must be able to decide if there are conflicts between operations that would compromise serializability of transactions. They must also know if an operation modifies the state of the transactional object. This information is encapsulated in the operation information hierarchy shown in figure 7.4.

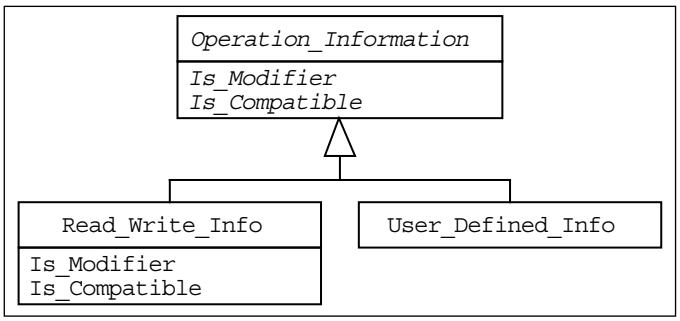

Figure 7.4: The *Operation_Information* Hierarchy

For each operation of a transactional object, an operation information object must be provided implementing the operations `Is_Modifier` and `Is_Compatible`, two functions that return a boolean value. `Is_Modifier` is needed for dealing with cooperative concurrency. It returns *true* if the operation modifies the state of the transactional object. This determines if mutual exclusion is needed when participants of the same transaction access the transactional object concurrently. `Is_Compatible` addresses competitive concurrency. The function must determine whether an operation conflicts with other operations available for this transactional object with respect to transaction serializability.

Strict concurrency control is provided in the OPTIMA framework by means of the concrete class `Read_Write_Information`. It implements classical strict concurrency control as shown in figure 7.2, allowing multiple readers or a single writer to access a transactional object at the same time.

Semantic-based concurrency control can not be provided out-of-the-box, since it depends on the properties of each transactional object. Nevertheless, the framework is flexible enough to allow a programmer of a transactional object to provide his or her own concurrency control by deriving from the abstract `Operation_Information` class and implementing the two required abstract functions. For commutativity-based concurrency control, the `Is_Compatible` operation must contain the compatibility table that allows to determine if the operation commutes with other operations of the transactional object[1].

At any given time, the concurrency manager associated with a transactional object holds the complete list of operation information objects representing all operation invocations made on behalf of transactions in progress. For a detailed description of how the

1. Depending on the update strategy chosen for a transactional object (see section 9.2.5 on page 115), forward or backward commutativity must be provided. If the framework has to support in-place *and* deferred update strategies for the same transactional object, then the `Operation_Information` class must be extended to provide two operations, `Is_Forward_Compatible` and `Is_Backward_Compatible`.

`Operation_Information` classes and `Concurrency_Manager` classes work together, see “Tying Things Together” on page 114.

Chapter 8:
Recovery

This chapter presents the detailed design of the recovery support for open multithreaded transactions. Recovery actions have to be taken in two situations: on transaction abort and in case of a system failure. Transaction abort can occur in the following situations:

- One of the participants explicitly aborts the transaction by raising an external exception.
- One of the participants can not handle a locally raised exception. The default handler will abort the transaction.
- A participant thread terminates without voting on the outcome of the transaction. This case is treated as an error (see section 4.4.4 on page 54).

A system failure can occur if:

- The process running the transaction support crashes.

It is the responsibility of the recovery support to handle all these cases, guaranteeing the *atomicity* and *durability* of all transactions at all time. In case of a system failure this implies more precisely that:

- If a transaction has committed before the failure occurred, then all modifications made on behalf of the transaction are reflected in the state of the transactional objects.
- All transactions that have not yet been committed successfully are aborted, which means that all modifications made on behalf of these transactions to transactional objects are undone, if necessary.

Note that the OPTIMA framework can only handle modifications made to transactional objects. Invocations of operations on non-transactional objects are ignored by the framework and can not be undone.

8.1 Global Design

In order to be able to recover from a system failure, the recovery support must keep track of the status of all running transactions and of the modifications that the participants have made to transactional objects on their behalf. This information, also called a transaction *trace*, must be stored on some kind of storage, called a *log*, that will not be affected by a system failure. Once the system restarts, the recovery support can consult the log and perform the cleanup actions necessary to restore the system to a consistent state. The information written to the log depends on the chosen recovery strategy, and the necessary cleanup actions depend on the strategy, the status of the transactions and whether the modifications made to the transactional objects have already been propagated from the cache to the non-volatile storage or not.

An overview of the recovery support architecture is given in figure 8.1. The *recovery management* component is the central unit. It controls the *persistence support*, the *cache manager* and the *log*.

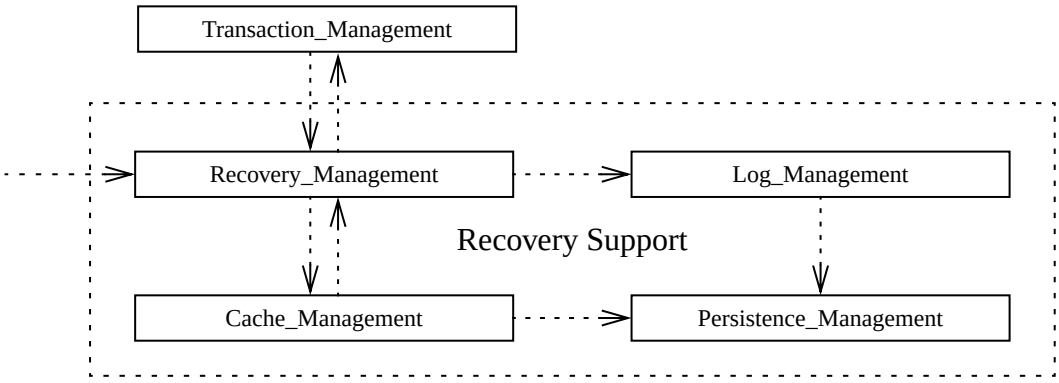

Figure 8.1: Recovery Support Overview

8.2 Persistence Support

This section presents the design of a persistence support that allows the state of any object, in our case transactional objects, to be written to any kind of storage device. The support has been designed to be general, since different forms of storage are needed throughout the transactional system. The following situations are examples of where the state of an object must be stored on some storage unit:

- The state of transactional objects must be stored on non-volatile storage in order to achieve *durability.*
- The log is stored on a special kind of non-volatile storage that must be able to survive system failures.
- Some recovery techniques make backup copies of the state of a transactional object in volatile memory (also called *checkpointing*). To undo modifications of the state of an object when a transaction aborts, the recovery support can simply replace the invalid state with the consistent state that was stored at the beginning of the transaction.

The persistence support is completely separated from the rest of the OPTIMA framework, and can therefore also be used in a stand-alone manner as presented in [KRS00]. The design of the persistence support strives to achieve the following goals:

- *Clear separation of concerns*: The object should not know about storage devices or about the data format that is used when writing the state of the object onto the storage unit and vice versa.
- *Modularity and extensibility*: It should be straightforward to define new persistent objects or add support for new kinds of storage.

8.2.1 Classification of Storage Devices

At some point, transactional objects must save their state on some storage device, so that it can be retrieved again in a later execution. The term storage is used in a wider sense here. Sending the state of the object over a network and storing it in the memory of some other computer would for instance also be valid, as long as the data survives program termination.

The kind of storage to be used for saving application data depends heavily on the application requirements. Properties such as performance, capacity of the storage media and particularities of usage (for instance *write-once* devices like CD writers) may affect the choice. Persistence can be implemented in a stronger form to support different kinds of fault tolerance, for instance for tolerating faults of the underlying hardware as required by transactions. To apply persistence properly in a fault-tolerant context, the application programmer has to identify the fault assumptions under which his or her system operates. Each storage device must provide documentation that allows a user to determine the reliability of the device. Based on this information, the application programmer can choose the device that fits the requirements of his or her application.

The OPTIMA framework organizes all supported storage devices in a class hierarchy as shown in figure 8.2. A concrete implementation of a storage class must derive from one of the abstract storage classes and implement the required operations. The `Storage` class represents the interface common to all storage devices. The operations `Read` and `Write` represent the operations that allow the user to read and write data from and to the storage device.

What kind of value types the operations must support will be discussed in more detail in the next subsection.

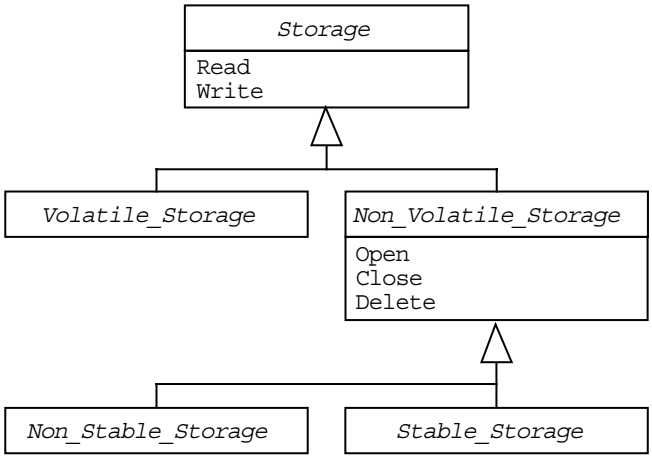

Figure 8.2: The *Storage* Hierarchy

At instantiation time, a transactional object is associated with a storage unit. Subsequently, the transactional object can save its state on the storage unit. To achieve decoupling, the *Strategy* design pattern described in section 5.2.2 on page 69 has been used. The *Strategy*, in our case the root storage class, declares the common interface to all concrete strategies. The *Context*, in our case the transactional object, uses this interface to make calls to a concrete storage class implementation defined by a *Concrete Strategy.*

Storage devices do not all have the same properties, and therefore must not all provide the same set of operations. The storage hierarchy is split into *volatile* storage and *non-volatile* storage. Data stored on volatile storage will not survive program termination. An example of volatile storage is conventional random access memory. Once an application terminates, its memory is usually freed by the operating system, and therefore any data still remaining in it is lost. Data stored on non-volatile storage, on the other hand, remains on the storage device even when a program terminates. Databases or disk files are common examples of non-volatile storage. Since the data will not be lost when the program terminates, additional housekeeping operations are needed to establish connections between the object and the actual storage unit, to cut off existing connections, and to delete data that will not be used anymore. These operations are `Open`, `Close` and `Delete`.

Even conventional memory can survive program termination if it is located in a process running on a remote machine. The framework comprises a parameterized class `Remote_Storage` that can turn any storage into non-volatile storage. The class implements the communication mechanism between the application and the process running on the remote machine. Invocations of storage operations on the local machine must be forwarded to the remote process, where they are executed on the actual storage object.

The design of the `Remote_Storage` class is also based on the *Strategy* design pattern. The type of storage that is used to store data on the remote machine can be chosen when

declaring the instance of the remote storage class. The `Remote_Storage` class is a descendant of the non-volatile storage class, since from the application point of view the remote storage device is non-volatile, even if a volatile storage is used on the remote machine.

In order to support fault tolerance, we distinguish between *stable* and *non-stable* storage devices among non-volatile storage devices. Data written to non-stable storage may get corrupted if the system fails in some way, for instance by crashing during the write operation. Stable storage ensures that stored data will never be corrupted, even in the presence of application crashes and other failures. The execution of the `Write` operation is atomic.

Stable storage has been first introduced in [LS79]. The paper describes how conventional disk storage that shows imperfections such as bad writes and decay can be transformed into stable storage, an ideal disk storage with no failures, using a technique called *mirroring*. When using this technique, data is stored *twice* on the disk (often two different physical disks are used to store the two copies of the data to increase reliability even more). If a crash occurs during the write operation of the first copy, the previously valid state can still be retrieved using the second copy. If the crash happens during the write operation of the second copy, the new state has already been saved in the first copy. When the system restarts later on, the state stored in the first copy must be duplicated and saved over the second copy. In order to decide which copy is valid, a third disk file called the *log* is used.

Using this mirroring technique, any non-volatile storage can be transformed into stable storage. Just as with the remote storage class, it is possible to write an implementation of the mirroring algorithm that is independent of the actual non-volatile storage class that will effectively be used to store the data. Again, the *Strategy* design pattern has been used to achieve this flexibility. The structure of the collaboration is shown in figure 8.3.

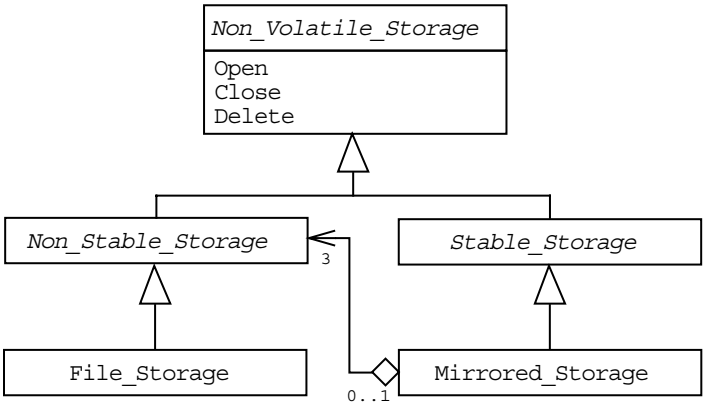

Figure 8.3: Stable Storage Based On Mirroring

When declaring a mirrored storage object, three non-volatile storage objects must be passed as a parameter to the constructor of the mirroring class. These could be, for instance, objects of the class `File_Storage` that implements storage based on the local file system. Using this technique, a variety of stable storage based on mirroring can be created reusing concrete implementations of non-volatile storage. The kind of non-volatile storage that will be cho-

sen depends on the needs of the application. To help the programmer, it is again very important that a concrete non-volatile storage implementation documents the assumptions under which the storage is considered non-volatile and other information that might be useful for the application programmer, such as performance. A detailed description of the design and implementation of the `Mirrored_Storage` class can be found in [CKS01].

The mirroring technique is not the only one that can be used to create stable storage. Database systems for instance have their own mechanism to guarantee atomic updates of data. Typically, they enclose updates of data in a database transaction. It is possible to write a concrete stable storage class that provides a bridge between an object-oriented programming language and a database.

Yet another way of providing stable storage is replication. The state of a transactional object can be broadcast over the network and stored on storage devices belonging to a set of remote machines. Remote memory provides very good performance. Although a replica can crash, the group of replicas as a whole can be considered stable; for as long as at least one of the remote machines remains accessible, the data can always be retrieved on a later execution.

Again, the replicated solution can be implemented in a generic way using the *Strategy* design pattern. The relationship between the context and the strategy is this time one to many. The `Replicated_Storage` class implements broadcasting and other replica management algorithms that handle failures of replicas during program execution. When declaring an instance of the replicated storage class, a storage object must be passed as a parameter to the constructor. Any type of storage can be used to store the data on the remote machine.

The complete storage class hierarchy and the relationships between the classes is shown in figure 8.4.

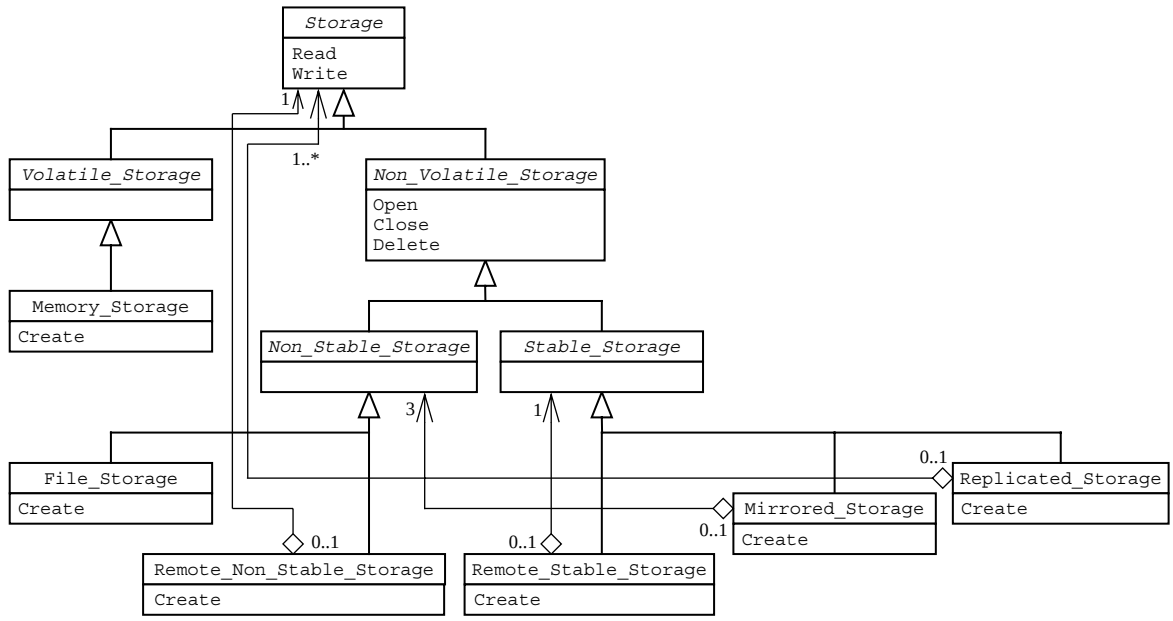

Figure 8.4: The Complete *Storage* Hierarchy

8.2.2 Object Serialization

When storing the state of a transactional object on some kind of storage unit, it must first be transformed from its representation in memory into some form that can be stored by the device. Most of the time the most convenient form will be a flat stream of bytes e.g. for storing data in flat files or sending data through network transport buffers. Interfaces to ODBMs can be more elaborate.

The *Serializer* design pattern, described in section 5.2.3 on page 70, is an ideal solution for this kind of problem. It provides a mechanism to efficiently stream objects into data structures of any form as well as create objects from such data structures.

The bigger the set of supported value types of the *Reader* / *Writer* interface is, the more type information can be used by the *Concrete Reader* / *Concrete Writer* to efficiently store the data on the backend. On the other hand, there are backends that support only a small set of value types. Flat files for instance only support byte transfer. For these kinds of backends the *Concrete Reader* / *Concrete Writer* must contain implementation code that maps the `read / write` operations of unsupported value types to the ones that are supported.

The big advantage of the *Serializer* pattern is that the application class itself needs no knowledge about the external representation format which is used to represent their instances. If this were not the case, introducing a new representation format or changing an old one would require to modify almost every class in the system.

In some object-oriented programming languages, such a serialization mechanism is already provided, which means that the `readFrom` / `writeTo` operations defined in the *Serializable* interface have predefined implementations for all value types of the programming language that are not covered by the *Reader* / *Writer* interface. The Java Serialization package [Sun98] or Ada streams [ISO95, 13.13] are examples of such predefined language support. If no language support is available, the `readFrom / writeTo` operations of the *Serializable* interface must be implemented for every *Concrete Element*.

8.2.3 Identification of Transactional Objects

Since the state of a transactional object survives program termination, there must be a unique way to identify a transactional object that remains valid during several executions of the same program. Also, when creating a transactional object, the user must be able to specify on what kind of storage he or she wants the state of the object to be saved. The object must be able to create an instance of the corresponding storage class and establish a connection to the storage device.

The information needed to create an instance of a concrete storage class is device dependent. To create a new file, a user must typically provide a file name that follows certain conventions, and maybe also a path name that specifies in which directory the file should be created. To access remote memory, an IP number or machine name must be provided. To solve this problem, a hierarchy of storage parameters has been introduced. This

hierarchy parallels that of the storage hierarchy and is shown in figure 8.5. Each storage device must define it's own storage parameter class that contains all the information needed to identify data stored on the device. Since a storage parameter object points to a unique location on a particular storage device, the storage parameter can also be used as a means for identifying a particular transactional object.

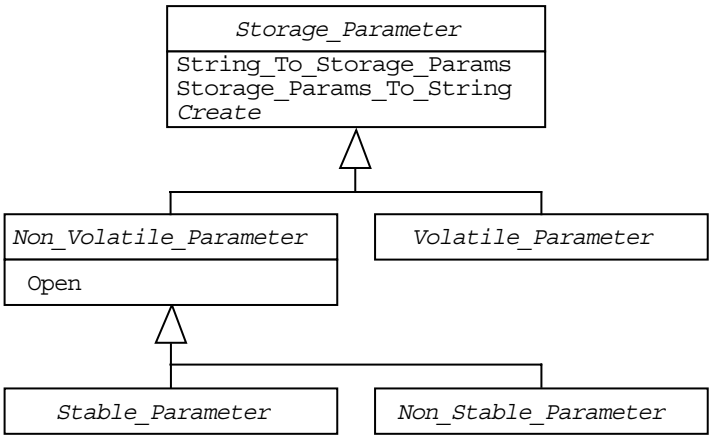

Figure 8.5: The *Storage_Parameter* Hierarchy

In order to allow the transactional object to instantiate a store object, the storage parameter class provides the `Create` method. The method instantiates the corresponding storage object, passing as a parameter the information stored inside the concrete storage parameter instance, and then physically creates the storage unit on the device. Non-volatile storage needs a second creator function, `Open`, that will instantiate the non-volatile storage class without creating a new storage unit on the device. Instead, a connection between the already existing data on the device and the storage object will be established. This technique is an instance of the well-known *Abstract Factory* design pattern described in section 5.1 on page 69. The `Create` and the `Open` methods define the connection between the two parallel class hierarchies.

Sometimes it can be convenient for a user to treat transactional objects in a uniform way. An object name in the form of a string has proven to be an elegant solution for uniform object identification [GJS96]. The two functions `Storage_Params_To_String` and `String_To_Storage_Params` provide a mapping between the two identification means.

8.2.4 Storage Management

Once a transactional object has been created and its state saved on a non-volatile storage device, it will theoretically remain on the device forever. The only way to remove the data and free the associated storage space is to explicitly delete the object. Forgetting to store the parameters that allow to identify the object on subsequent application runs can result in permanent storage leaks. This can be prevented by hard-coding the parameters in the application code, or by storing the parameters in some other transactional object. In a sense,

storage parameters act as persistent pointers to transactional objects. All transactional objects inside an application must be reachable by following pointers stored inside transactional objects whose parameters are statically known.

In order to create a simple flat hierarchy of transactional objects, an application programmer can use the transactional directory class provided by the framework. When creating a new transactional object, the storage parameters must be registered with this directory. At any time, the application programmer can then consult the list of existing transactional objects stored in the directory and determine which of them is still needed and which of them can be deleted. The creation of a transactional object and the updating of this directory must be atomic, or else again storage leaks can occur. This is why these two actions must be performed inside the same transaction. The same reasoning applies to the deletion of transactional objects.

When saving the state of the directory, the storage parameters of all transactional objects that have been created in the system must be written to the associated storage device. It is therefore important that all storage parameter classes implement the *Serializable* interface.

8.3 Caching Support

In order to improve the performance of the overall system, the states of transactional objects are kept in main memory. Accessing memory is in general significantly faster than accessing the non-volatile storage devices that are used for permanently storing the states of transactional objects. However, in systems that are composed of lots of transactional objects, it is often not possible to keep the state of all objects in memory at a given time. This is why such systems usually use a cache that only keeps a subset of the objects in memory.

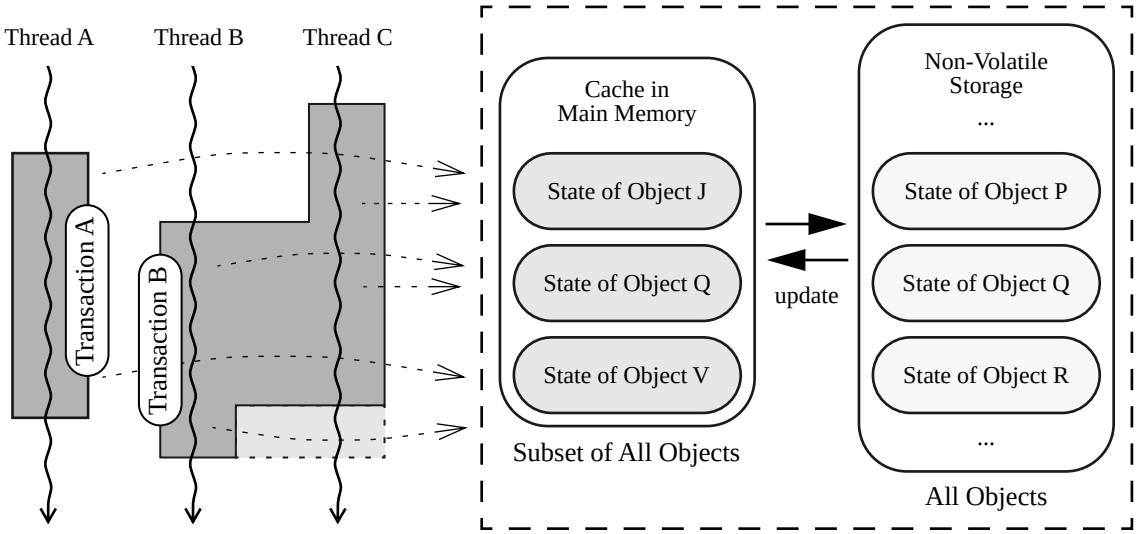

Figure 8.6: Caching for Transactional Objects

As shown in figure 8.6, the existence of the cache is completely transparent to the threads executing the transactions. When an operation is invoked on an object for the first time, the state of the object is loaded from the associated storage unit into the cache. Subsequent accesses to the object can now be serviced a lot faster. If an operation modifies the state of an object, the object is marked as being *dirty.*

In practice, caches are very effective because of the *principle of locality,* which is an empirical observation that, most of the time, data in use is either the same data that was recently in use (*temporal locality*), or is data "nearby" the data recently used (*spatial locality*). The behavior of caches can be tailored in order to get a better hit ratio, i.e. by adjusting the size of the cache, or by choosing appropriate fetch and replacement algorithms.

8.3.1 Cache Fetch Algorithm

In general, objects are loaded into the cache on demand, i.e. when an operation is invoked on the object. If the object is not present in the cache, the calling thread must wait until the cache loads the state of the object from the associated non-volatile storage device. As an alternative, the cache can prefetch objects by guessing which objects are likely to be accessed in the future.

8.3.2 Cache Replacement Algorithm

When an object's state is brought into the cache, it is often necessary to delete the state of an object that is already in the cache due to the lack of space. In a conventional cache, the state of any object can be replaced. This is not true for caches used in a transaction system. Firstly, we distinguish between *Steal* and *No-Steal* policy. In the *Steal* policy, objects modified by a transaction in progress may be propagated to the associated storage unit at any time, whereas in the *No-Steal* policy, modified objects are kept in the cache at least until the commitment of the modifying transaction. We also make a distinction on what happens during transaction commit. In the *Force* policy, all objects that a transaction has modified are propagated to their associated storage units during the commit process, whereas in the *No-Force* policy no propagation is initiated upon transaction commit.

Depending on the application requirements, different replacement strategies can be appropriate. A well known replacement strategy is *Least-Recently-Used* (LRU), where for each cached object the cache keeps track of *when* it was used the last time. When there is not enough space for a new object to be loaded into memory, the state of the least recently used object is written to the storage unit, and then the in-memory state of the object is discarded.

8.3.3 Extensible Cache Design

In order to make caching possible, the framework must store important information for each data object, e.g. if the state of the object is currently in memory or on the associated non-volatile storage unit. When an operation is invoked on the data object and the object's state is not currently in memory, the state must be loaded from the associated storage unit. This is the main task of the `Memory_Object` class shown in figure 8.7.

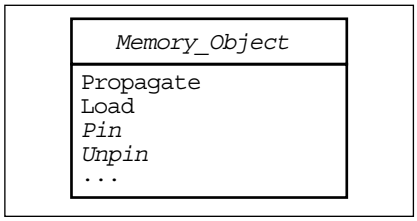

Figure 8.7: The Memory Object

Invoking `Propagate` instructs the memory object to save the associated data object to the corresponding storage unit. The `Load` method can be used to re-initialize the state of the data object with a previously saved state. Calling `Pin` will "pin" the data object in memory, meaning that its state can not be propagated to the associated storage unit, until `Unpin` is called. The memory object class has additional methods related to concurrency control and recovery. They are presented in section 9.2.5 on page 115.

Each instance of the memory object class handles the state of one application object. The cache manager controls all these instances, replacing application objects in memory by propagating them to the associated storage unit according to the cache policy. Defining an optimal cache policy depends on the application requirements. It is therefore important for the framework to allow a user to define his or her own cache policy. This flexibility can again be achieved using the *Strategy* design pattern.

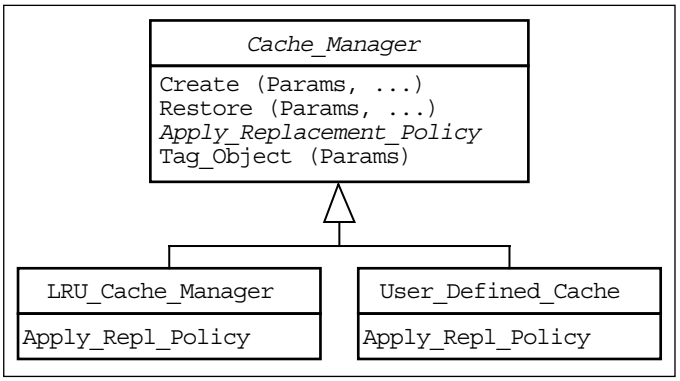

Figure 8.8: The *Cache_Manager* Hierarchy

The abstract root class `Cache_Manager` defines the operations that must be provided by a concrete cache implementation as shown in figure 8.8. `Create` and `Restore` are two methods that are called when a user instantiates a transactional object. `Create` will physically create a new object on the storage unit identified by the storage parameter `Params`, whereas `Restore` attempts to initialize the object's state with a previously saved state. If there is not enough memory available to allocate space for a new application object, then the abstract method `Apply_Replacement_Policy` is invoked. This results in freeing memory by writing the state of application objects to their associated storage unit. Which objects are chosen depends on the replacement strategy. The method `Tag_Object` is called during recovery to tell the cache manager that the state of the transactional object identified by the `Params` parameter must be recovered prior to any further use of the object.

8.3.4 Consequences of Caching

Although introducing a cache is completely transparent for the users of the transaction support, it significantly complicates the reasoning about the consistency of the state of the system. When using a cache, the current state of a transactional object is determined by the state of the object in the cache, or if it is not present in the cache, by the state of the object on the associated storage device. When a transaction aborts, the state changes made on behalf of the transaction are undone in the cache. These changes might have already been propagated to the storage unit. Fortunately, we do not have to undo them, since they will be undone the next time we update the state of the object on the storage device. When a transaction commits, we must ensure that at some time in the future, the changes of the transaction will be propagated to the associated storage device.

Using a cache has a significant impact on the actions that must be taken when recovering from a crash failure. On a system crash, the content of the cache is lost, and therefore, in general, the state of the objects on their associated storage devices can be inconsistent for the following reasons:

- The storage unit does not contain updates of committed transactions.
- The storage unit contains updates of uncommitted transactions.

When recovering from a system crash, these situations must be remedied. The former problem can be solved by *redoing* the changes made by the corresponding committed transactions, the latter by *undoing* the changes made by the corresponding aborted transactions. These two techniques are explained in detail in section 8.5.

8.4 Logging

The system log is a sequential storage area located on stable storage (see section 8.2.1). It is important that the log is stored on stable storage, since it must always remain readable even in the presence of failures in order to guarantee the properties of transactions. The purpose of the log is to store information necessary to reconstruct a consistent state of the system in case a transaction aborts or a system crash occurs. The required information can be split into five categories:

- Undo Information
- Redo Information
- Creation Information
- Deletion Information
- Transaction Status Information

There are three situations in which the log is updated:

- A transaction is committed or aborted.
- A transactional object is created or deleted.
- An operation that modifies the state of a transactional object is invoked.

Undo and redo information can be stored in the log in two ways. In the first technique, called *physical logging*, copies of the state of a transactional object are stored in the log. These copies are called *before-images* or *after-images*, depending on if the snapshot of the state of the object has been taken before or after invoking the operation. Unfortunately, physical logging only works if strict concurrency control is used (see “Strict Concurrency Control” on page 84). If semantic-based concurrency control such as commutative locking is used, undo and redo information must be saved using *logical logging*. In this technique, the operation invocations and their parameters are written to the log. In order to support undo, every operation `op` of a transactional object must provide an *inverse operation* op^{-1}, i.e. an operation that undoes the effects of calling `op`.

After a system crash, the entire log needs to be scanned in order to perform all the required redo and undo actions. The kind of algorithm to be used to recover from a crash depends on the chosen recovery strategy and is detailed in the next subsection.

A log could potentially grow to be very long, making recovery prohibitively slow. This problem can be solved by *checkpointing* the log, which forces all the updated objects in the cache to be propagated to their associated storage devices. After checkpointing, the log entries pertaining to committed and aborted transactions that precede the checkpoint are obsolete and can be garbage collected. There have been many different proposals for checkpointing a transaction log, with different performance trade-offs between recovery processing and checkpoint processing, which of course delay normal processing. An overview of checkpointing mechanisms and how they relate to the different recovery strategies is given in [BHG87].

8.4.1 Encapsulating Logging Techniques

Physical and logical logging techniques are captured in the class hierarchy presented in figure 8.9.

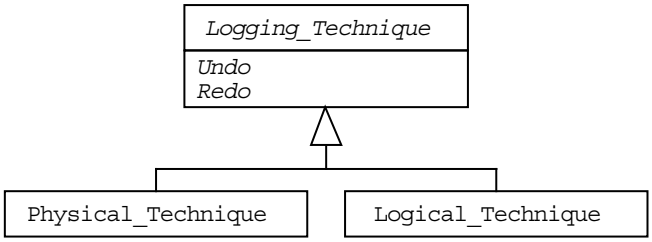

Figure 8.9: The *Logging_Technique* Hierarchy

As mentioned before, commutativity-based concurrency control requires logical logging to be used. Transactional objects using strict concurrency control are free to choose between physical and logical logging. In this case it is mainly a performance issue. If the size of the state of a transactional object is small, it might be more efficient to save the entire state of the object after an operation has been invoked than saving the operation together with its parameters. Physical logging is also more efficient in situations where a single transaction performs several calls to operations of a transactional object that update its state. In that case a *single* before-image saved to the log allows the changes of all operations to be undone, whereas logical logging requires all operation invocations and their parameters to be saved to the log.

8.4.2 Encapsulating Log Information

As mentioned before, five different kinds of information must be stored in the log. Again, this information can conveniently be encapsulated in objects that are organized in a class hierarchy as shown in figure 8.10. As a result, the log only contains objects of this hierarchy, which facilitates the recovery processing in case of a system failure.

Any kind of information stored in the log belongs to a transaction. For that reason the root-level class has an attribute that stores the transaction identifier. The `Transaction_Information` subclass is used to store the state changes of transactions in the log. The remaining information classes all concern objects. The object is identified using a storage parameter, stored in an attribute of the `Object_Information` class. The `Undo_Information` and `Redo_Information` class derive from this class. They store undo respectively redo information using a physical or logical logging technique object introduced in section 8.4.1. The two classes `Creation_Information` and `Deletion_Information` provide means for undoing creation or deletion of transactional objects in case a transaction aborts. The `Deletion_Information` class uses a physical logging technique object to store the last state of the destroyed transactional object.

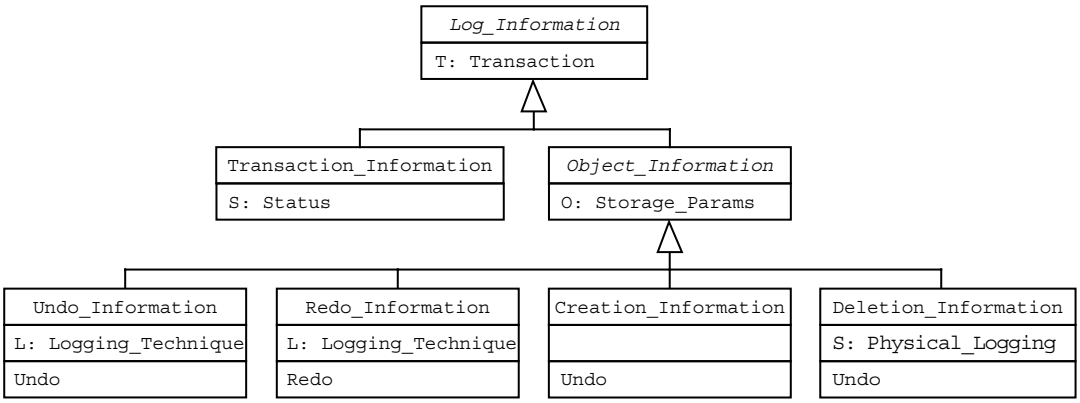

Figure 8.10: The *Log_Information* hierarchy

8.5 Recovery Support

The recovery support of a transaction system must ensure that the *atomicity* and *durability* properties of open multithreaded transactions are satisfied, even in the presence of system failures. As mentioned above this can be achieved by using recovery strategies based on *undo* operations, *redo* operations, or both.

8.5.1 Recovery Strategies

8.5.1.1 Undo/Redo

The *Undo/Redo* recovery strategy requires both undo and redo actions for every operation on transactional objects. This strategy allows great flexibility in the management of the cache by permitting *Steal* and *No-Force* object replacement policies. It maximizes efficiency during normal operation at the expense of less efficient recovery processing.

8.5.1.2 Undo/No-Redo

The *Undo/No-Redo* recovery strategy requires undo but never uses redo actions because it ensures that all the updates of committed transactions are reflected on the storage devices associated with the transactional objects. It therefore relies on *Steal* and *Force* cache replacement policies. The commitment of a transaction is delayed until all its updates are recorded on the storage units associated with the transactional objects involved in the transaction. If there happens to be a system failure during this propagation phase causing the transaction to abort, the previous state can be reconstructed by undoing the corresponding operations.

8.5.1.3 No-Undo/Redo

The *No-Undo/Redo* recovery strategy, also known as *logging with deferred updates*, never requires undo actions, but relies on redo actions. Updates of active transactions are not propagated to the storage units associated with transactional objects, but recorded in the system log, either in the form of an after-image of the state if physical logging is used, or as a list of invoked operations, also called an *intention list*, if logical logging is used. If a crash occurs after a transaction has committed, the lost state can be reconstructed by redoing the corresponding operations. No-Undo/Redo recovery relies on *No-Steal* and *No-Force* cache replacement policies.

8.5.1.4 No-Undo/No-Redo

The *No-Undo/No-Redo* recovery strategy avoids undo actions by creating private shadow versions of each modified object on stable storage (see section 8.2.1). Redo actions are avoided by atomically replacing the actual objects in stable storage with their corresponding shadow versions associated with the transaction during transaction commit. This, of course, requires a special form of stable storage that provides a *commit-and-replace* operation. The states of all transactional objects must be saved on this kind of storage.

8.5.2 Encapsulating Recovery Strategies

The OPTIMA framework supports *Undo/NoRedo*, *NoUndo/Redo* and *Undo/Redo* recovery managers. *NoUndo/NoRedo* recovery is not supported, for it requires a special form of stable storage that is difficult to implement based on general non-volatile storage. Again, the different recovery managers are organized using a class hierarchy as represented in figure 8.11.

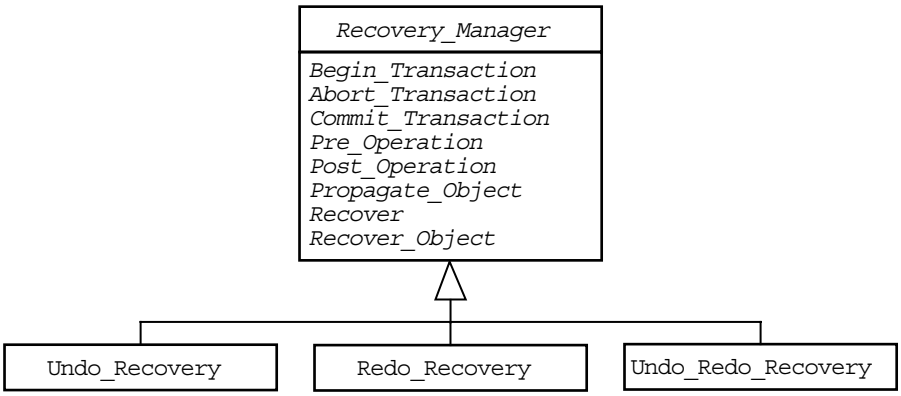

Figure 8.11: The *Recovery_Manager* Hierarchy

All recovery managers provide the same interface as shown in the abstract root class `Recovery_Manager`. A general description of the operation is given here. Detailed descriptions of the algorithms used for each recovery strategy are presented in the next subsections.

- `Begin_Transaction`, `Abort_Transaction` and `Commit_Transaction` are called by the transaction support when the status of a transaction changes. This status change must be written to the log. Depending on the recovery strategy, the recovery manager uses different algorithms for aborting or committing transactions.
- `Pre_Operation` and `Post_Operation` are called before respectively after every method invocation on a transactional object. Depending on the recovery strategy used, the recovery manager must gather undo and redo information.
- `Propagate_Object` is called by the cache manager when it wants to write the state of a transactional object to its associated storage unit in order to free up memory. Depend-

ing on the recovery strategy, certain actions, e.g. writing undo information to the log, must be performed before the object's state may be deleted from memory.

- `Recover` is called during startup of the system if the system has not been shut down properly during the previous run. The recovery manager must scan the log and take the necessary actions to recover a consistent application state. The algorithm used depends on the recovery strategy. Every object whose state needs to be recovered is marked as such by calling the `Tag_Object` method of the cache manager.
- `Recover_Object` is called by the cache manager when a participant wants to access an object whose state must be recovered.

The recovery manager must write information to the log in order to be able to perform correct recovery in the presence of system failure. The information needed to perform recovery depends on the recovery strategy.

The following general rules must always be followed [BCF+97]:

- *Undo Recovery Rule* (or *Write Ahead Logging*):
  All information necessary for undoing the changes made to the state of an object during execution of one of its operations must be written to the log before the object's state is propagated to its associated storage unit.
- *Redo Recovery Rule* (or *Commit Rule*):
  All information necessary for redoing the changes made to the states of all objects modified during a transaction must be written to the log before the transaction commits.

The following subsections describe the implementation of the recovery manager methods for the different recovery strategies by means of pseudo code. Note that although Undo/Redo recovery uses very complicated algorithms, commercial systems that need to process large amounts of data often use this technique, because it allows the states of transactional objects to be written back to their associated storage unit at any time.

8.5.3 Undo/NoRedo Recovery Algorithms

`Pre_Operation`

- Pin the transactional object
- If physical logging is used
  - If this is the first operation modifying the state of this object invoked during this transaction
    - Save before-image in memory
- If logical logging is used
  - Save undo information for this operation in memory

`Post_Operation`

- Unpin the transactional object

`Propagate_Object`

- Write all undo information for this object from memory to the log

`Abort_Transaction`

- For all modified objects
  - Undo the changes to the object in memory by applying the gathered undo information in memory
  - Optional: Propagate the state of the object to the associated storage unit
  - Discard undo information for this transaction
  - Send *abort* notification to the concurrency control of the object
- Log transaction abort

`Commit_Transaction`

- For all modified objects
  - If the object is dirty
    - Propagate the objects state to the associated storage unit
- Log transaction commit
- For all modified objects
  - Send *commit* notification to the concurrency control of the object
  - If we are in a top-level transaction
    - Delete undo information for this transaction
  - Else
    - Pass undo information of the object to parent transaction

8.5.4 NoUndo/Redo Recovery Algorithms

`Pre_Operation`

- Pin the transactional object
- Optional: Save undo information in memory

`Post_Operation`

- If physical logging is used
  - If this is the first operation modifying the state of this object invoked during this transaction
    - Save after-image in memory

    - Else
        - Replace previous after-image with new one
- If logical logging is used
    - Save redo information for this operation in memory
- Do not unpin the transactional object

`Propagate_Object`

- Can not be called inside a transaction, since the object stays pinned

`Abort_Transaction`

- Log transaction abortion
- For all modified objects
    - Undo the changes to the object in memory[1]
    - Delete redo information for this transaction
    - Unpin the object
    - Send *abort* notification to the concurrency control of the object

`Commit_Transaction`

- If we are in a top-level transaction
    - For all accessed objects
        - Write the redo information to the log
- Log transaction commit
- For all accessed objects
    - Send *commit* notification to the concurrency control of the object
    - If we are in a top-level transaction
        - Unpin the object
        - Optional: propagate the objects state to the storage unit
    - Else
        - Pass redo information of the object to parent transaction

8.5.5 Undo/Redo Recovery Algorithms

`Pre_Operation`

- Pin the transactional object
- If physical logging is used
    - If this is the first operation modifying the state of this object invoked during this transaction

1. This can be done for instance by using the saved state on the associated storage unit, or by applying undo operations saved in memory during the `Pre_Operation` operation.

        - Save before-image in memory
- If logical logging is used
    - Save undo information for this operation in memory

`Post_Operation`

- If physical logging is used
    - If this is the first operation modifying the state of this object invoked during this transaction
        - Save after-image in memory
    - Else
        - Replace previous after-image with new one
- If logical logging is used
    - Save redo information for this operation in memory
- Unpin the transactional object

`Propagate_Object`

- Write all undo information for this object from memory to the log

`Abort_Transaction`

- Log transaction abortion
- For all modified objects
    - Undo the changes to the object in memory
    - Discard redo information for this transaction
    - Discard undo information for this transaction
    - Unpin the object
    - Send *abort* notification to the concurrency control of the object

`Commit_Transaction`

- If we are in a top-level transaction
    - For all accessed objects
        - Write the redo information to the log
- Log transaction commit
- For all accessed objects
    - Send *commit* notification to the concurrency control of the object
    - If we are not in a top-level transaction
        - Pass redo information of the object to parent transaction

Chapter 9:
Interfacing with Programming Languages

The previous three chapters have detailed the design of the three main components of the framework, namely the transaction support, the concurrency control and the recovery support. This chapter presents how these components work together, and what must be considered when designing an interface to the transaction framework for the application programmer. The elegance of this interface depends on the features available in the particular programming language.

In order to correctly handle transactions, the transaction support and recovery support components must be notified in the following situations:

- When modifying the state of an open multithreaded transaction (e.g. when starting, joining, closing, aborting or committing a transaction), and
- Before and after every method invocation on a transactional object.

When designing an interface for the application programmer, transaction identification management and calls to the transaction support should be automated and hidden as much as possible. A good interface will also force the programmer to adhere to the rules that govern open multithreaded transactions.

9.1 Associating Participants with a Transaction

For every new transaction, a new transaction context is created. In our framework, the transaction context is encapsulated in an instance of the transaction class (see section 6.5 on page 77).

Once a thread becomes a participant of a transaction, the transaction context must be associated with it. When it subsequently invokes methods on transactional objects or when it votes on the outcome of the transaction, the transaction support must be able to determine on behalf of which transaction the participant is working.

For this reason, traditional transaction systems usually define a *transaction identifier* type, *TID* for short. When a new transaction is started, a unique transaction identifier instance is created, associated with the transaction context and handed back to the thread that started the transaction. Subsequent method invocations on transactional objects or calls to the transaction support must pass this transaction identifier as a parameter, in order to identify the transaction on behalf of which the operation is to be executed.

Of course, this approach is very cumbersome and annoying for the application programmer. The approach is even error-prone, for it does not enforce one of the basic rules for open multithreaded transactions, namely that a thread can only participate in one transaction at a time. With explicit transaction identifiers, a “malicious” transaction programmer can perform with one thread work for two transactions simultaneously by using two different transaction identifiers when invoking methods on transactional objects. This violates the isolation property, since results from one transaction might be “smuggled” to the other one.

Fortunately, most modern concurrent object-oriented programming languages allow a system programmer to associate data with threads. This can be done e.g. in Java by extending the thread class and adding attributes. A way of achieving this effect in Ada 95 is presented in section 10.3.3 on page 135.

Using such a technique, the transaction identifier can be associated with a thread that starts or joins a transaction. The transaction support is then able to verify that a given thread only participates in one transaction at a time, and thus information smuggling is impossible. In addition, transaction nesting can be handled automatically: a participant of a transaction that starts a new transaction will automatically start a nested transaction.

9.2 Encapsulating Objects

It would be convenient for the application programmer to be able to work with transactional objects just like with any other kind of object. This illusion can be achieved by writing a wrapper object, subsequently called the *transactional object*, that encapsulates the real object, i.e. the *data object*.

Ideally, an application programmer should be able to use any data object inside a transaction. In particular, it should be possible to reuse data objects that have been written for non-transactional applications, by “magically” transforming them into transactional objects.

A transactional object must provide a certain set of functionalities. They are summarized below:

Operations called by the Application Programmer

- The transactional object must provide all operations that the original object provides.
- Since transactional objects are durable, the transactional object must provide operations that allow the application programmer to create new instances of the object, to restore instances created previously, and to delete obsolete instances.

Operations called by the Components of the Transaction Framework

- The transactional object must provide operations for saving and loading the object's state to / from the associated storage unit.
- For each operation on the original data object, the transactional object must provide concurrency control and recovery information, classifying each operation as *observer* or *modifier*. If semantic concurrency control is used, a compatibility table relating all operations on the original data object must be provided.
- If logical logging is used, the transactional object must designate undo operations for each operation on the original object, and provide operations for loading and saving operation invocations.

9.2.1 The Transactional Object

In order to hide all this complexity from the application programmer, each data object is hidden behind a wrapper object, the *transactional object*, that offers the same interface as the original data object, plus operations for creating, restoring and deleting the transactional object. The `Create` and `Restore` operations use a storage parameter (see section 8.2.3 on page 95) to identify the storage unit on which the state of the transactional object is to be stored.

When an operation is invoked on the transactional object, it must be able to determine on behalf of which transaction the invoking thread is currently working. If the programming language provides a mechanism that allows the transaction support to associate a transaction identifier with a thread, the transaction context can be passed to the transactional object in a transparent manner. Otherwise every operation of the transactional object must have an additional parameter `TID` that identifies the transaction on behalf of which the operation is to be executed.

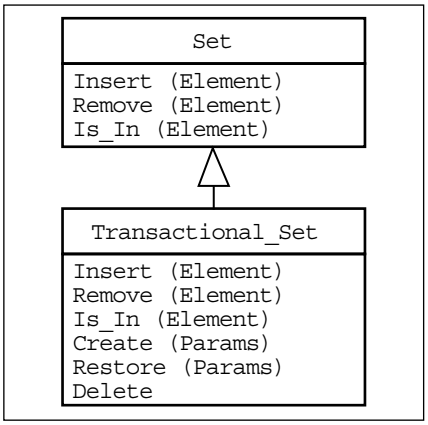

Figure 9.1: A Transactional Set

In order to be compatible with the original data object, the transactional object may inherit from or implement the same interface as the original data object[1]. As a consequence, algorithms that work with original objects will also accept transactional objects[2]. Figure 9.1 shows an example of a `Set` data object and its corresponding `Transactional_Set` object.

Parts of the functionality of a transactional object are common to all transactional objects, e.g. that they must call the recovery manager before and after every operation invocation. Other functions are specific to a particular transactional object, e.g. what operation can be invoked on the original data object.

The common functionalities are implemented by the memory object class (see section 8.3.3 on page 99), the specific functionalities must be implemented for each transactional object separately.

9.2.2 Handling Durability

Transactional objects must be capable of loading and saving their state from and to a storage unit. The classes `Loading_Operation` and `Saving_Operation` shown in figure 9.2 encapsulate this behavior. The `Saving_Operation` for instance has a single operation `Save` with two arguments, the original data object and the storage (see section 8.2.1 on page 91) on which the state must be saved. How the state of a data object must be saved to a storage unit depends on the specific data object, and must therefore be implemented in the transactional object by deriving from the `Saving_Operation` class and implementing the `Save` operation for the specific data object. The same applies for the `Loading_Operation` class.

Creation and deletion of a transactional object also depend on the specific data object, for they might require complex memory management. This behavior is encapsulated in the classes `Creation_Operation` and `Deletion_Operation` shown in figure 9.2.

9.2.3 Encapsulating Operation Invocations on Data Objects

The `Normal_Operation` shown in figure 9.2 encapsulates operation invocations on a data object. It has four abstract operations, namely `Do_Operation`, `Undo_Operation`, `Get_Operation_Info` and `Is_Update`. For each operation of the original data object, an operation class must be implemented that derives from the `Normal_Operation` class and implements the required operations. An instance of this class is used during execution to encapsulate calls to the associated operation of the original data object. If logical logging is used, the state of the instance of this class representing the operation invocation is written to the log. It must therefore store the *in* and *out* parameters of the operation.

The `Insert_Element` operation of our *Set* abstract data type for instance has one input parameter, namely the element to be inserted. Hence, the operation class for the

1. This does not work if the programming language does not allow to associate the transactional context with a thread. In that case, the transaction identifier must be passed explicitly to each operation of the transactional object, and hence the interface for the transactional object will differ from the original interface.
2. Of course this technique only works if all objects that are used in transactions are created by the application programmer. If a method of a data object A creates a data object B, then the method of the transactional object A should create a transactional object B. If the creation is done based on the *Factory* design pattern, then the problem can be solved by replacing the data object B factory by a transactional object B factory. Otherwise, the method of object A must be rewritten.

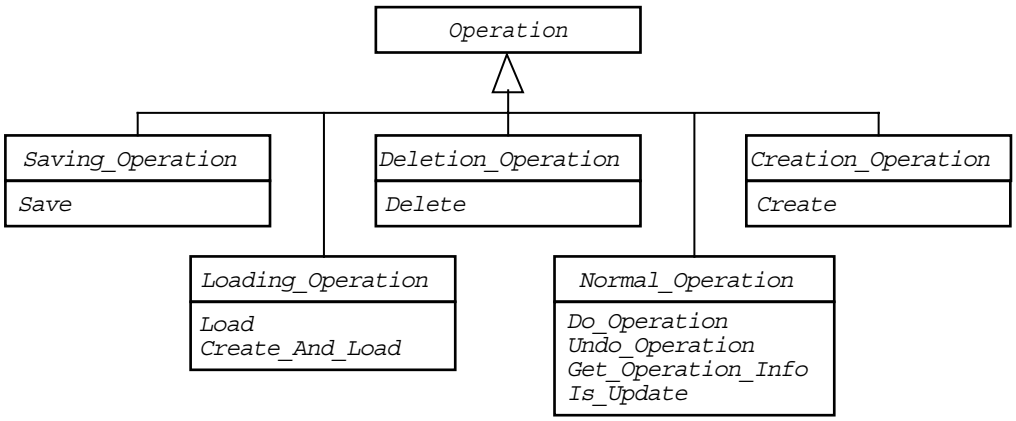

Figure 9.2: The *Operation* Hierarchy

`Insert_Element` operation must define an attribute that can store the element passed as a parameter to the method call. This attribute is called `To_Insert` in figure 9.3.

As a consequence, `Do_Operation` is simple to implement. A call to `Do_Operation` must invoke the operation on the original data object. In our case, invoking `Do_Operation` of the `Insert_Operation` class will call the `Insert_Element` operation on the original set object, passing as a parameter the element stored in the attribute `To_Insert`.

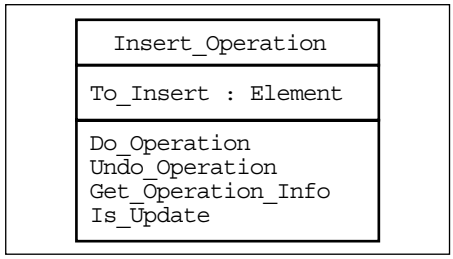

Figure 9.3: An Example Operation

The `Undo_Operation` must only be implemented if logical logging is used. In the set example, the undo operation is also easy to implement, for it corresponds to invoking the `Remove_Element` operation on the original data set, passing as a parameter the element that has been previously inserted.

`Get_Operation_Info` is the operation that encapsulates the concurrency control information for the operation. It must simply return an instance of the `Operation_Information` class hierarchy (see section 7.4 on page 83). If strict concurrency control is used, then the predefined `Read_Write_Information` class can be returned. For semantic-based concurrency control, an application-class-specific operation information class must be implemented by deriving from the abstract `Operation_Information` class.

The last operation, `Is_Update`, is called by the framework to determine if an invocation of the operation changes the durable state of the transactional object. If this is true, then the transaction support knows that it must save the new state of the object to the associated storage unit upon transaction commit.

The necessary calls to the concurrency control and recovery support components to be performed when executing an operation on a transactional object do not depend on the transactional object itself. They can therefore be written once and for all, and reused in all transactional objects. The class `Atomic_Call` encapsulates concurrency control and recovery processing in a single operation named `Atomic_Do`. It executes the following actions:

1. *Concurrency Control Prologue*: Call the `Pre_Operation` method (see section 7.3 on page 82) of the concurrency control manager associated with the transactional object, passing as an argument the instance derived from the `Normal_Operation` class that represents the operation to be executed. This action allows the concurrency control manager to perform the actions necessary for handling competitive and cooperative concurrency.
2. *Recovery Prologue*: Call the `Pre_Operation` method (see section 8.5.2 on page 104) of the recovery manager, which informs the transaction support that the transactional object has been accessed by calling the `Add_Transactional_Object` method of the transaction class (see section 6.5 on page 77). Then, all operations needed for providing correct recovery according to the chosen recovery strategy are performed (see section 8.5.2 on page 104).
3. *Execute the Operation*: Invoke the operation on the original data object by executing the `Do_Operation` method of the `Normal_Operation` class.
4. *Recovery Epilogue*: Call the `Post_Operation` method of the recovery manager, which again might perform operations needed for recovery, such as storing redo information.
5. *Concurrency Control Epilogue*: Call the `Post_Operation` method of the concurrency control manager associated with the transactional object, informing it of the successful completion of the operation invocation. This action allows the concurrency manager to perform necessary clean-up operations, such as discarding the cooperative concurrency control information.

9.2.4 Tying Things Together

The interaction of all the previously described “helper” objects is also independent of the actual data object, and can therefore be written once and for all and reused by all transactional objects. The `Memory_Object` class introduced in section 8.3 on page 97 already encapsulates accesses to the original data object, since it is the class that manages memory and storage usage. All operation invocations on the data object are done through the memory object. It therefore is the perfect candidate for managing the concurrency control and recovery information for the data object.

Figure 9.4 illustrates the relations between all helper objects that are created for each data object. The `Transactional_Object` class defines and contains instances of all auxiliary classes that are specific for a particular data object, namely the `Loading_Operation`, `Saving_Operation`, `Creation_Operation`, `Deletion_Operation`, and all `Normal_Operation` classes. In addition, it contains a reference to the memory object that encapsulates accesses to the data object.

The memory object contains the storage unit that is associated with the data object, the concurrency manager for the data object, and it contains a reference to the original data

object, if it is currently in memory. The memory object makes use of the operation objects defined in the transactional object in order to perform operations on the data object.

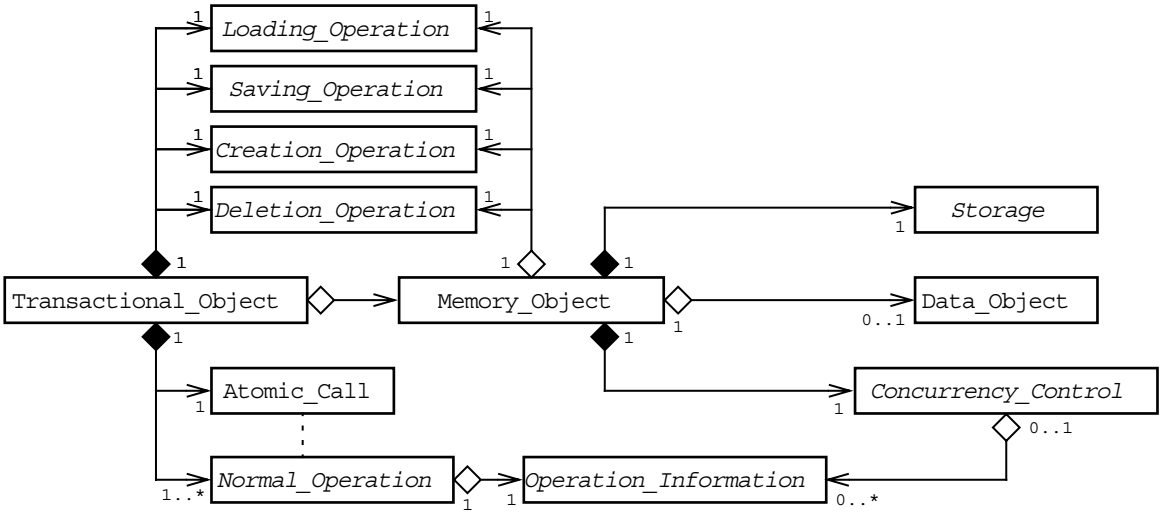

Figure 9.4: Encapsulation of a Data Object

9.2.5 In-place Update and Deferred Update

There exist two different strategies for performing updates on the state of a data object, namely *in-place update* and *deferred update* [KS99].

When using *in-place update*, an operation invocation forwarded from the transactional object to the `Memory_Object` is directly executed on the data object. The effects of the operation invocation are therefore visible to all following operation invocations, even those made on behalf of other transactions[1].

When using *deferred update*, operations are executed on copies of the data object. All operations executed on behalf of the same transaction are invoked on the same copy of the data object, and therefore the resulting changes are not visible to other transactions. Only upon transaction commit, the effects of the operations are applied to the original data object, either by copying the state changes from the copy to the original data object, or by reapplying the operations executed within the transaction on the original data object.

For each transactional object an update strategy must be chosen. The update strategy determines the kind of commutativity information that must be provided in case semantic-based concurrency control is used. In-place update requires backward commutativity, whereas deferred update requires forward commutativity (see section 7.4.2.1 on page 85).

1. When using pessimistic concurrency control, conflicting operations are caught during the concurrency control prologue, and therefore will never get to this point.

The two update strategies are implemented in subclasses of the Memory_Object class as shown in figure 9.5.

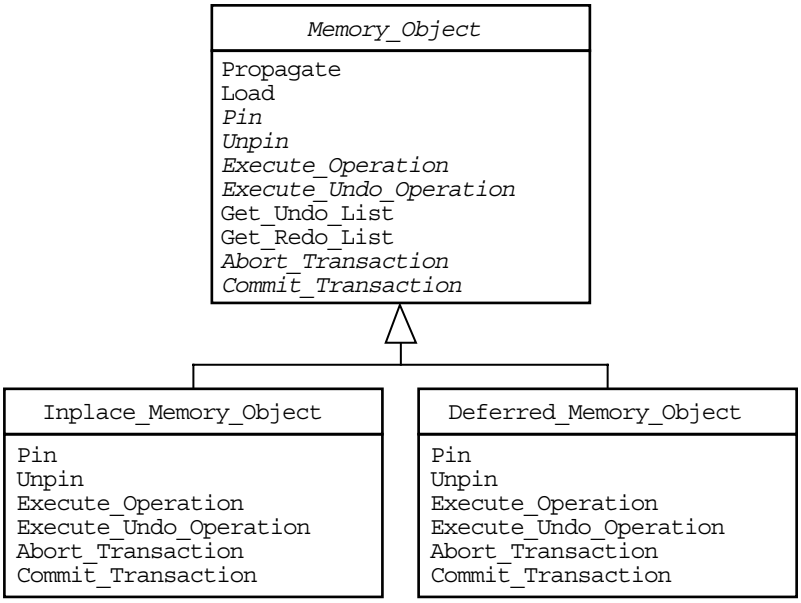

Figure 9.5: The *Memory_Object* Hierarchy

In addition to the four operations already presented in section 8.3.3 on page 99, figure 9.5 also shows the other operations that the Memory_Object class provides to the recovery manager and the Atomic_Call class.

Execute_Operation first collects undo information depending on the update strategy and logging technique, and then executes the operation on the data object (or on a copy). Before returning, redo information is collected, again depending on the update strategy and logging technique. Execute_Undo_Operation executes the inverse operation.

Get_Undo_List or Get_Redo_List are called by the recovery manager. These functions return a list of Logging_Technique objects (see section 8.4.1 on page 101) that the recovery manager can save to the log, e.g. when preparing for transaction commit.

Abort_Transaction and Commit_Transaction are called by the recovery manager to inform the memory object about the outcome of a transaction. As a consequence, the collected undo and redo information is used to update the state of the original data object, depending on the update strategy. In case of abortion or commitment of a top-level transaction, the undo and redo information can subsequently be discarded.

9.2.6 Trace of an Operation Invocation

To clarify how the objects presented in figure 9.4 work together, this section describes an operation invocation in more detail. Figure 9.6 shows a sequence diagram of what exactly happens when invoking the Insert_Element operation of the Transactional_Set introduced in figure 9.1.

The Concurrency_Control, Recovery_Manager, Transaction, Memory_Object and Atomic_Call objects are independent of the particular data object that is to be encapsulated, and can therefore be reused for different data objects. The data object itself, here the Set object, and the associated transactional object Transactional_Set must be written for each data object. The objects that encapsulate method invocations and operation information, here Insert_Operation and Insert_Operation_Info, must be written for each method of the original data object.

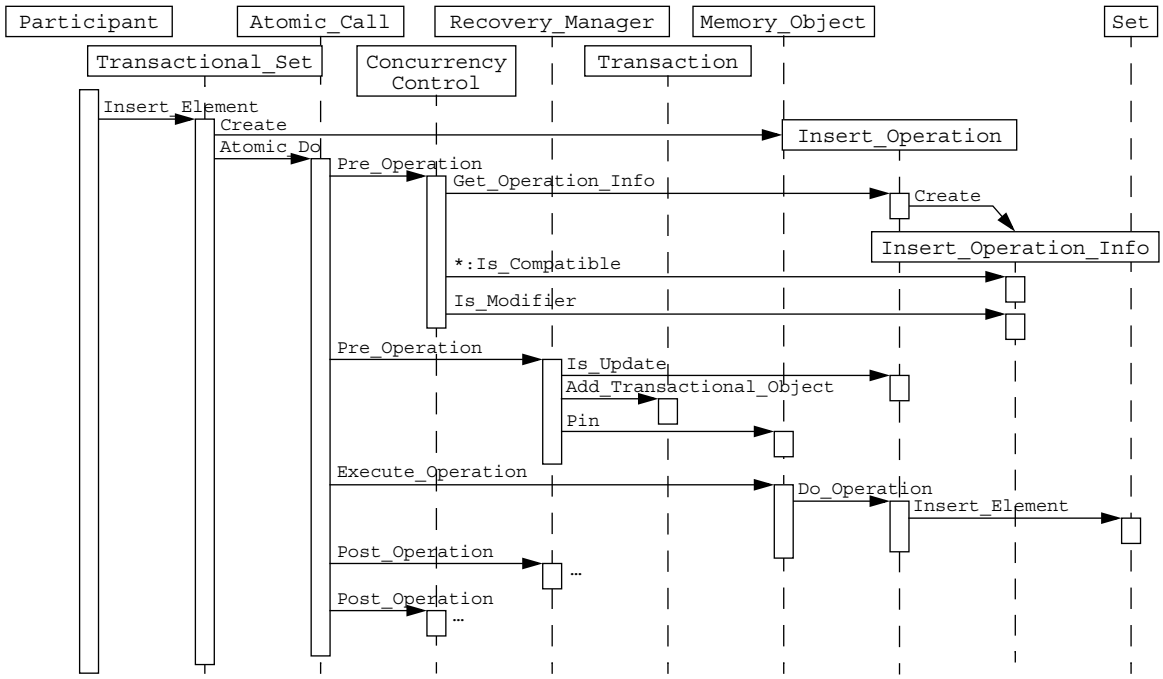

Figure 9.6: An Operation Invocation on a Transactional Object

The invocation begins when a participant thread invokes the Insert_Element method on the Transactional_Set object. In our example, the transaction on behalf of which the method is to be executed is transparently linked to the thread as described in section 9.1.

Inside the Insert operation of the Transactional_Set object, an instance of an Insert_Operation is created and the element to be inserted stored in one of its attributes. Then, the Atomic_Do method of the Atomic_Call object is called, passing the Insert_Operation as a parameter. Atomic_Do starts by calling the Pre_Operation method of the Concurrency_Control object associated with the Transactional_Set. The Concurrency_Control first asks the Insert_Operation for its operation information, which results in creating an instance of the Insert_Operation_Info class, and then, in the case of pessimistic concurrency control, repeatedly calls Is_Compatible to ensure that the operation invocation does not conflict with any operation invocation made by transactions in progress. This phase corresponds to handling competitive concurrency, i.e. isolation of transactions. If this first phase completes successfully, then the Is_Modifier method of the

`Insert_Operation_Info` is called in order to determine if the `Insert_Element` method modifies the state of the `Set` object or not. If this is the case, then the mutual exclusion lock associated with the transactional set must be acquired. This second phase handles cooperative concurrency.

Next, the `Pre_Operation` method of the `Recovery_Manager` is invoked. It asks the `Insert_Operation` if the `Insert_Element` method modifies the durable state of the `Set` object, in which case the object is added to the list of modified objects by calling `Add_Transactional_Object` of the `Transaction` object associated with the calling thread. Then, the real data object is asked to be kept in memory by calling the `Pin` method of the `Memory_Object`. This might require using the `Loading_Operation` to load the state of the object from its associated storage unit.

Finally, the `Execute_Operation` method of the `Memory_Object` is called. At this point, depending on the recovery strategy, undo information must be gathered. Then, the `Do_Operation` method of the `Insert_Operation` object is invoked. There, the element to be inserted that had been stored in one of its attributes is extracted and the actual call to `Insert_Element` is executed on the `Set` data object.

After the call, again depending on the recovery strategy, redo information is gathered, and the `Post_Operation` of the `Recovery_Manager` is invoked to execute the recovery epilogue. Finally, the `Post_Operation` of the `Concurrency_Control` is called, which will release the mutual exclusion lock held on the object. Of course, the competitive concurrency information can not be discarded. It must be kept until the outcome of the transaction has been determined.

9.3 Initializing and Shutting Down the Transaction Support

An interface must provide a way to customize the framework to the application needs. In particular, the programmer should be able to chose what kind of cache manager and what kind of recovery manager should be used. Moreover, there should be ways to specify on what kind of storage the log shall be stored.

This can be done by providing a simple operation or equivalent construct that lets the programmer specify his or her choices. Using C or Java-like syntax such an operation may look as follows:

```
void system_init (cache_manager *my_cache_manager,
                  recover_manager *my_recovery_manager,
                  storage_params my_log_storage_params);
```

Likewise, if ever the system must be brought down, a special operation should be called. It writes all dirty transactional objects to their associated storage units, and performs all outstanding log updates.

9.4 Providing Transactions at the Programming Language Level

Depending on the features available in the programing language, different interfaces for transactions are possible.

9.4.1 Procedural Interface

The most commonly used interface to transactions is the *procedural interface*.

For supporting the open multithreaded transaction model we need five operations. For each operation, the version with a transaction identifier and the one without a transaction identifier are shown. A C++ or Java-like syntax is used.

Starting an Open Multithreaded Transaction

- With TID:
```
tid begin_transaction();
tid begin_transaction(unsigned participants);
tid begin_transaction(tid parent);
tid begin_transaction(tid parent, unsigned participants);
```
- Without TID:
```
void begin_transaction();
void begin_transaction(unsigned participants);
```

Calling this operation starts a new transaction, creating a new transaction object by instantiating the `Transaction` class and invoking its `Begin_Transaction` method (see section 6.5 on page 77). If TID management must be preformed by the application programmer, the TID of the newly created transaction must be handed back as a result of the operation call. In this case we also need a separate operation for starting a nested transaction, requiring the application programmer to pass the parent transaction identifier as a parameter. If TID management can be hidden from the application programmer, there is no need to differentiate starting a new transaction and starting a nested transaction. The transaction support can examine the transaction context of the calling thread, and if it is already associated with a transaction, a nested transaction is created.

In both cases, an additional parameter can be supplied to the `begin_transaction` procedure that specifies the maximum number of participants that the transaction will accept. Once this number is reached, the transaction is closed automatically.

Joining an Open Multithreaded Transaction

- With TID:
```
void join_transaction(tid transaction);
```
- Without TID:
```
void join_transaction(thread *participant);
```

This procedure allows a thread to join an ongoing open multithreaded transaction. The parameter `transaction` specifies which transaction the thread wants to join. With TID management, the transaction to be joined is identified by means of a transaction identifier, with-

out TID management, the transaction to be joined can be identified by passing a reference to one of its participants. In that case, a check is made to verify that the calling thread is either a participant of the parent transaction, or is not participating in any transaction at all.

Once the transaction support knows which transaction to join, it calls the `Join_Transaction` method of the corresponding transaction object.

Starting or Joining Named Transactions

In some situations it can be convenient to associate a name with an open multithreaded transaction. As a result, all threads that want to participate in a common transaction do not have to obtain a tid or a reference to one of its participants before being able to join the transaction. In this case, there is even no need to distinguish between starting or joining a transaction.

- With TID: `tid begin_or_join_transaction(string s);`
- Without TID: `void begin_or_join_transaction(string s);`

When the first thread executes the operation, the transaction support will start a new transaction and associate the transaction with the name provided as a parameter; subsequent threads calling the operation with the same name will join the transaction.

Closing an Open Multithreaded Transaction

- With TID: `void close_transaction(tid transaction);`
- Without TID: `void close_transaction();`

By calling this procedure, a participant can close the transaction, making it impossible for subsequent threads to join. If hidden TID management is used, the transaction support can verify that the calling thread really is a participant of the transaction. To close the transaction, the transaction support calls the `Close_Transaction` method of the corresponding transaction object.

Committing an Open Multithreaded Transaction

- With TID: `void commit_transaction(tid transaction);`
- Without TID: `void commit_transaction();`

This procedure must be called by a participant that wants to commit the changes that it has made on behalf of a transaction. Again, the TID-less version allows the transaction support to verify that the calling thread is really a participant of the transaction. Then, the transaction support invokes the `Commit_Transaction` method of the corresponding transaction object, which will block the calling thread until the outcome of the transaction has been determined. If some other participant votes abort, then the transaction is aborted and the exception `Transaction_Abort` is raised.

Aborting an Open Multithreaded Transaction

- With TID: `void abort_transaction(tid transaction);`
- Without TID: `void abort_transaction();`

Calling this procedure aborts the transaction. Just as with the close and commit procedures, the transaction support can verify that the calling thread has previously started or joined a transaction when TID-less management is used. Then, the `Abort_Transaction` method of the corresponding transaction object is invoked. Aborting a transaction does not block the calling thread. Any participants that have been previously suspended while attempting to commit the transaction are released.

9.4.1.1 Discussion

Of course, the version of the procedural interface that does not expose TID management is much more secure, since it prohibits information smuggling between transactions.

The procedural interface is very flexible, but unfortunately, it also has some serious drawbacks.

It is, for instance, possible to start or join a transaction, but forget to vote on its outcome, which results in blocking all other participants that behave correctly. But what is even more annoying is that the procedural interface can not catch unhandled exceptions crossing the transaction boundary and abort the transaction as required.

To still achieve the desired effect, the programmer must adhere to certain programming conventions that depend on the programming language. The conventions that must be followed for the Ada 95 programming language are presented in section 9.4.1 on page 119.

9.4.2 Object-Based Interface

The trick of the *object-based interface* consists in associating the lifetime of a transaction with the lifetime of an object or a group of objects in case of multiple participants.

The object-based interface declares a `Transaction` class as shown in figure 9.7. Threads wishing to start a transaction do so by declaring an object of that class.

```
class Transaction {

  Transaction();
  Transaction(thread *participant);
  Transaction(String s);
  ~Transaction();

  public void close();
  public void commit();
}
```

Figure 9.7: Object-Based Interface

There are three constructors. Calling the one without parameter will start a new transaction or a new nested transaction. If a reference to a thread is passed as a parameter to the constructor, then the transaction support attempts to join the current transaction of the referenced thread. The third constructor provides for starting or joining a named transaction.

Closing and committing a transaction can be achieved by calling the corresponding methods of the declared transaction object.

There is no need to provide an abort method. When the object goes out of scope, the destructor is invoked automatically. If the participant has not previously called the commit method, then the transaction will be aborted. The advantages of using this technique are three-fold. Firstly, every participant of a transaction is guaranteed to vote on the outcome of the transaction. If the application programmer forgets to call the commit method of the transaction object, then, following a safe approach, the transaction is aborted. Secondly, unhandled exceptions automatically cause the transaction to abort, because the destructor of the transaction object is invoked when the block in which the object has been declared is left. Finally, deserters, i.e. threads disappearing without voting on the outcome of a transaction, can also be detected using the same mechanism, resulting in aborting the transaction.

Section 11.2.4 on page 163 presents the object-based interface for the Ada programming language, and section 13.3.3.1 on page 205 and section 13.3.3.2 on page 207 show how it has been used in the auction system example.

9.4.2.1 Discussion

Clearly, the object-based interface is more elegant than the procedural interface, and it enforces the rules of open multithreaded transactions by construct. The structure of a program reflects the structure of transactions.

This rigid structure, however, is not always appropriate. Event-driven systems, e.g. user interfaces, often use callbacks that contain the statements to be executed in case a certain event occurs. The object-based interface does not lend itself to implement a transaction that encapsulates the execution of multiple callbacks, or a transaction that is started inside one callback, and committed inside some other callback[1].

On top of that, the object-based interface is in this form only feasible in a stack-based programming language, where the lifetime of an object is associated with a block structure. In heap-based languages, or languages that use garbage collection as e.g. Java, objects are not necessarily destroyed when a block is left. Often, such programming languages do not provide the possibility to declare destructors at all.

9.4.3 Object-Oriented Interface

When using the object-oriented interface, a whole open multithreaded transaction is encapsulated inside an object representing the transaction. The work to be performed by an individual participant of the transaction is encapsulated in a method of this programmer-defined object.

1. Work-arounds are of course always possible.

The object-oriented interface defines an abstract `Transaction` class with methods for beginning or joining, closing, committing and aborting the transaction. These methods are protected methods, which means that they can only be called from within the class hierarchy.

To define a concrete open multithreaded transaction, the application programmer must derive from the `Transaction` class, and add public methods for each type of participant.

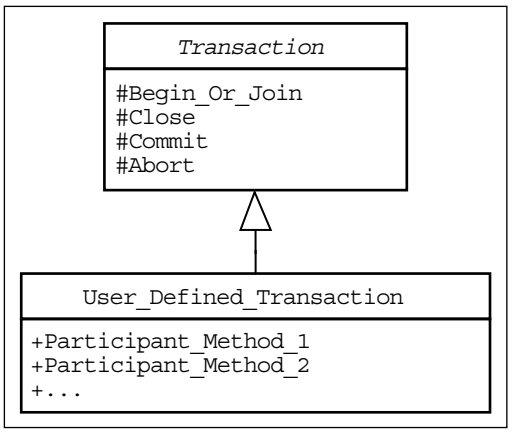

Figure 9.8: Object-Oriented Interface

Just as in the procedural interface, the application programmer must follow certain programming guidelines. At the beginning of each participant method, the `Begin_Or_Join` method must be invoked. In this situation, the identity of the `Transaction` object can be used to identify the transaction instead of a name. At the end of the execution of the method, `Commit` or `Abort` must be called. In order to correctly handle exceptions, the method must contain an exception handler for each internal exception, and a default exception handler that calls `Abort` to catch unhandled internal exceptions.

Of course, it is also possible to apply the technique presented in the object-based interface section for each participant method. Instead of calling `Begin_Or_Join` and `Commit` or `Abort`, each participant method must declare a participant object, whose constructor and destructor take care of the necessary calls to the transaction support.

The advantage of the object-oriented interface is that the entire open multithreaded transaction, i.e. the program code for each participant, is grouped together inside a class. This clearly improves readability, understandability and maintainability of the transaction as a whole. Code reuse is also possible, for transactions that want to perform similar work can derive from some other transaction class, override or add new participant methods, and reuse old ones.

If the programming language provides a syntax to associate exceptions with methods like in Java, the interface of the transaction class can also clearly state the possible external exceptions that might be raised by each participant method.

The object-oriented interface that has been developed for Ada is shown in section 11.2.5 on page 166.

9.5 Additional Considerations

Recently, two new concepts have attracted attention within the community of object-oriented researchers and practitioners, namely *reflection* and *aspect-oriented programming*.

These concepts focus on transparency and separation of concerns, and may be helpful when integrating the transaction framework with a programming language.

The last two sections of this chapter introduce both concepts and give hints on how these mechanisms can be applied in order to provide even smoother transaction interfaces for the application programmer.

9.5.1 Reflection

Reflection is a general methodology for describing, controlling, and adapting the behavior of a computational system. When using reflection, the important static or dynamic execution characteristics and parameters of a system are made concrete in one or more programs that represent the default computational behavior. This description or control program is called a *metaprogram*. By specializing parts of this metaprogram, a programmer can customize the execution of the application program, changing execution strategies, mechanisms and data representations.

The concept of reflection can be quite naturally applied to the object-oriented world. Just as objects in conventional object-oriented programming are representations of real world entities, they can themselves be represented by other objects, usually referred to as *meta-objects* [Mae87], whose computation is intended to observe, modify and control the behavior of their *referents*, i.e. the objects they represent. This idea can be applied recursively, i.e. meta-objects can be represented by meta-meta-objects, and so on. A reflective system is thus structured in multiple levels, constituting a *reflective tower*. The objects at the base level are termed *base-objects* and perform computations on the entities of the application domain. The objects at the meta-levels perform computations on the objects residing at the lower levels.

The association between base-objects and meta-objects can be many to many, i.e. several meta-objects may share a single referent, and a single meta-object may have multiple referents. The interface between adjacent levels in the reflective tower is usually termed a *meta-object protocol* [KdB91], MOP for short.

Meta-computation is often implemented by intercepting the normal computation of objects. In other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either substituting or encapsulating the referent's action. *Transparency* is an important concept of reflection. The objects at each level are completely unaware of the presence and workings of the objects at the levels above.

9.5.1.1 Applying Reflection

Reflection has been successfully applied to separate *functional* from possibly multiple *non-functional* features in the design and implementation of a system. In a typical approach, the base-level objects of a system implement the application's functional requirements, while meta-objects augment the base-level functionality ensuring non-functional properties.

Reflection has been used in existing systems to control distribution, fault tolerance and communication [GGM94], persistence [LZ94], atomicity [SW95], and authentication [ACF99].

[BP95] describes how reflection techniques have been used for incorporating extended transaction models such as split and joint transactions (see section 3.4.5 on page 38) into an existing commercial transaction processing system. In their approach, the programming interface to transactions is separated into different levels, where each level represents a different view of the transaction functionality.

The first level, called the *transaction demarcation interface*, provides the operations `begin-E-transaction`, `commit-E-transaction` and `abort-E-transaction`. The second level, the *extended transaction interface*, provides operations specific to each extended transaction model. For the split and joint transaction model this level contains operations such as `split-transaction` and `join-transaction`. Level 3, the actual meta-level, extends the implementation of the transaction processing monitor to support the extended transaction interface defined in level 2 by defining operations such as `instantiate`, `reflect`, `delegate-up`, `delegate-lock`, `form-dependency`, and `no-conflict`. By extending the meta-objects at level 3, the application programmer can tailor the behavior of the underlying transaction processing monitor to fit the chosen extended transaction model.

9.5.2 Aspect-Oriented Programming

Aspect-oriented programming [KLM+97] has emerged based on the observation that object-oriented programming techniques fail to clearly express design decisions that cross-cut the structure chosen to provide a system's functionality. Such cross-cutting concerns may include error handling, synchronizing concurrent entities, data sharing, memory allocation, minimizing network traffic, data representation, distribution, etc.

Aspect-oriented programming makes a clear distinction between *components*, which represent the properties of the system for which the implementation can be cleanly encapsulated behind a well-defined interface, and *aspects*, which are cross-cutting properties that affect the performance and run-time behavior of the components in systematic ways.

To better support the expression of cross-cutting design decisions, aspect-oriented programming uses a component language to describe the basic functionality of the system, and aspect languages to describe the different cross-cutting properties. Designing an aspect-oriented programming system requires to well understand what must go into the component language, what must go into the aspect languages, and what must be shared among the languages. The component language must allow the programmer to write component programs that implement the system’s functionality, while at the same time ensuring that those programs don’t pre-empt anything the aspect programs need to control. The aspect languages must support implementation of the desired aspects, in a natural and concise way. The com-

ponent and aspect languages usually provide different abstraction and composition mechanisms.

A special language processor called the *aspect weaver* is used to coordinate the composition of aspects and components. Essential to the function of the aspect weaver is the concept of *join points*, which are those elements of the component language semantics that the aspect programs coordinate with. Aspect weavers work by generating a *join point representation* of the component program, and by then compiling the aspect programs with respect to it. The join point representation can also be generated at run-time using a reflective run-time for the component language.

9.5.3 Evaluation

Both reflection and aspect-oriented programming techniques can be used to improve the integration of the transaction framework with a programming language.

A straight-forward idea is to use these techniques to transparently transform data objects into transactional objects. This approach seems promising, since [LZ94] demonstrates for example how to use reflection to transparently add persistence to objects. But it is not enough to be able to add persistence, or recoverability, or concurrency control to an object. What needs to be investigated is how these aspects interact with each other when applied together. It is not clear for instance how to avoid inconsistent combinations, such as choosing physical logging, but commutativity-based concurrency control. Also, it seems impossible to determine the compatibility table for the operations of a transactional object without the help of the application programmer.

Aspect-oriented programming and of course reflection can also be used to simplify the interface to transactions. A possible approach is to associate methods and transactions by executing an entire method of an object inside a transaction.

Using AspectJ [KHH+01], an aspect-oriented programming environment for Java, a programmer can for instance write a *transactional aspect*. The aspect can be defined in a way that when applying it to a method of a Java class, it will add a call to `begin_transaction` at the beginning of the method, and a call to `commit_transaction` at the end of the method call. Additionally, the entire method is enclosed in a `try - catch` block to catch unhandled exceptions and to abort the transaction if necessary.

Although this approach is very simple, it is also dangerous when applied to methods that have not been designed to be executed inside a transaction. Consider the following simple example, where a shared object is used to synchronize two threads. The object provides two methods, `store_result1` and `store_result2`. Two threads in the system calculate each a result, and store it in the shared object by calling one of the `store_result` methods. The `store_result` method stores the result in the corresponding attribute of the class, and suspends the calling thread until both results have been calculated.

If the `store_result` methods are executed normally, the program works perfectly well. The first thread that has computed the result will be blocked until the second thread has completed its calculation.

The situation changes if the methods are executed as transactions. The problem arises due to the inter-transaction concurrency control implementing the isolation property of transactions. Even though a thread is suspended, the resources obtained on behalf of the transaction can not be released, since the outcome of the transaction has not been determined yet. Depending on the interleaving of the operations, either the second thread can not write its result, since the other thread is holding the rights to access the attribute, or both threads can write their results, but no one can consult the other's result to determine if waiting is necessary. In both cases, the resulting deadlock can not be resolved by aborting one of the transactions.

The point here is that synchronization between threads is actually some form of cooperation, and must be performed inside the same transaction. The problem can therefore be solved by specifying that both `store_result` methods are to be executed inside the same open multithreaded transaction.

In larger systems, collaboration between threads can be difficult to survey, and it is not trivial to guarantee that adding transactional semantics to method calls does not create potential for deadlocks.

Nevertheless, reflection and aspect-oriented programming open interesting perspectives to integrate the transaction framework with a programming language, and provide elegant means for customizing it to the application needs. There was no time for further research in this direction, but there are plans to investigate interfaces based on these techniques in the Java implementation of the framework (section 14.2.4 on page 217).

Part III

Implementation for Ada 95

Chapter 10:
Ada 95

Ada 95 is a revised and much improved version of the “classical” Ada programming language developed originally for the United States Department of Defense to match their requirements for a modern, safe, and efficient structured programming language. Classical Ada was codified as an ANSI standard in 1983 and is therefore sometimes called “Ada 83”; an equivalent ISO standard was ratified in 1987. The successor language Ada 95 is defined by the ISO standard ISO/IEC 8652:1995 [ISO95]. For an overview of the history of Ada, see the rationale [Bar97].

In order to define an elegant interface between a programming language and the OPTIMA framework, the way the programming language addresses the fundamental concepts presented in chapter 2 must be analyzed.

10.1 Ada 83 vs. Ada 95

Ada 95 improves over the original definition of Ada 83 in several areas, including the following:

- Addition of language constructs for object–oriented and incremental application development (“tagged types” and “child packages”).
- Improvements in the area of tasking: a new construct, called “protected type”, implements passive entities that may be accessed concurrently by several tasks.
- A distribution model based on remote procedure calls is standardized.

For a complete description of all the new features in Ada 95, see the language standard [ISO95] and the rationale [Bar97].

10.2 Object–Oriented Programming in Ada

Object–oriented programming in Ada 95 is based on the concept of derivation classes formed by type extension [Wir88]. Type extension works by refining an existing record type by adding new components or operations, or by modifying existing operations. Contrary to e.g. Oberon–2 [MW91], not all record types can be extended: only *tagged* types can. This language design choice was motivated mainly by a concern for efficiency.

The *tagged type* represents what in general is called a *class* in object-oriented terms (see section 2.1.6 on page 12). *Methods* of a tagged type are called *primitive operations* in Ada terms. A tagged type is a record type defined with the keyword `tagged`, or a type derived from such a tagged record type. Figure 10.1 shows a simple tagged type with a few extensions. The derived types inherit all their ancestors' components and primitive operations. New components may be added with each derivation, inherited primitive operations may be overridden if the inherited behavior is not appropriate for the derived type, and new primitive operations may be added. Ada 95 only supports *single inheritance*.

```
with Canvases; use Canvases;
package Shapes is

  type Shape is
    abstract tagged null record;

  procedure Draw
    (S      : in     Shape;
     Canvas : access Canvas'Class)
    is abstract;

end Shapes;

package Shapes.Circles is

  type Circle is new Shape with
    record
      -- Added components
      Center : Point;
      Radius : Float;
    end record;

  procedure Draw
    (C      : in     Circle;
     Canvas : access Canvas'Class);
  -- Inherited abstract primitive
  -- operation, must be overridden.

  function Radius
    (C : in Circle) return Float;
  -- New primitive operation;

end Shapes.Circles;

package Shapes.Rectangles is

  type Rectangle is new Shape with
    record
      Top_Left     : Point;
      Bottom_Right : Point;
    end record;

  procedure Draw
    (R      : in     Rectangle;
     Canvas : access Canvas'Class);
  -- Inherited and overridden.

  function Width
    (R : in Rectangle) return Float;
  -- New primitive operation;

end Shapes.Rectangles;
```

Figure 10.1: A Tagged Type Hierarchy

A particularity of the object–oriented concepts of Ada 95 is class–wide programming. It makes explicit the dynamic polymorphism that in other object–oriented languages often is inherent. For each tagged type `T` there is a corresponding class–wide type `T'Class`, comprising the whole tree of types derived directly or indirectly from `T`, including `T` itself. Values of such a class–wide type can hold any value of any of the types derived from `T`. In particular, access–to–class–wide types may be used to reference any value of any type in the derivation class, enabling a programmer to implement e.g. heterogeneous collections.

Class–wide types are also used in dispatching calls, i.e. calls to primitive operations where the target operation is determined *at run time* depending on the dynamic type of the value of a class–wide type, i.e. depending on its tag. This is different from most other object–oriented programming languages, where either all method invocations are dispatching, e.g. in Java, or where it must be specified at the declaration of the method whether or not calls will be dispatching, e.g. the `virtual` methods in C++. In Ada 95, the programmer can decide at each call whether or not it should dispatch. If the controlling operand, i.e. the actual parameter, of a call to a primitive operation has a class–wide type, the call is dispatched, otherwise, the primitive operation of the type of the controlling operand is invoked. The difference is illustrated by the code fragment in figure 10.2.

```
type Shape_Ref is
  access all Shape'Class;

...
```

```
declare
  A_Shape  : Shape_Ref := ...;
  A_Canvas : aliased Canvas;
begin
  Draw (A_Shape.all,
        A_Canvas'Access);
  -- This call dispatches on the
  -- actual type of the object
  -- 'A_Shape' references -- this
  -- might be either 'Circle' or
  -- 'Rectangle' in this example.
end;
```

```
declare
  A_Circle : Circle := ...;
  A_Canvas : aliased Canvas;
begin
  Draw (A_Circle, A_Canvas'Access);
  -- This call is not dispatching:
  -- the actual parameter is not of
  -- a class-wide type.
end;
```

Figure 10.2: Illustrating Dispatching Calls

Dispatching in Ada 95 is *safe* — there always exists a primitive operation to dispatch to at run time; it is not possible to write a program that would dispatch to a non–existing method as it may happen for example in Smalltalk: the language rules do not allow this, and hence such errors will be caught by the compiler.

Dispatching occurs only on primitive operations; it is never controlled by a formal parameter having a class–wide type. Such subprograms are called *class–wide* operations. Any actual parameter that has a type in the given derivation class may be passed in place of a formal class–wide parameter.

Finally, Ada 95 provides *abstract* tagged types that may have abstract primitive operations. An abstract type defines characteristics common to all types derived from it; its primitive operations, be they abstract or concrete, are inherited and form a specification that all types derived from it have to adhere to. An abstract type may also have concrete primitive operations — this is useful to express default behavior for all types derived from the abstract type. A concrete type derived from an abstract type must supply implementations for the inherited abstract operations.

10.2.1 Controlled Types

Controlled types [ISO95, 7.6] have been introduced in Ada 95 to facilitate resource management within abstract data types while preserving the abstraction. To make a type controlled, it must be derived from one of two standard abstract tagged types, `Ada.Finalization.Controlled` or `Ada.Finalization.Limited_Controlled`. The controlled type then inherits the following operations from these root types:

- `Initialize` — is automatically invoked whenever an object of the controlled type is created without explicit initialization.
- `Finalize` — is called just before a controlled object is destroyed, i.e. goes out of scope, is deallocated using an instantiation of `Ada.Unchecked_Deallocation`, or is overwritten during an assignment.
- `Adjust` — is invoked for the target object in an assignment just after it has been overwritten. The type `Limited_Controlled` has of course no primitive operation `Adjust`, since objects of limited types can not be copied.

The default implementations of these three primitive operations do nothing at all. By overriding the inherited versions in the derived type, the application developer can precisely control object creation, destruction, and assignment. This can be used for instance to implement automatic storage management for an abstraction at the application level without having to clutter its interface.

10.3 Concurrency in Ada

10.3.1 Tasks

According to the classification introduced in section 2.2.2 on page 16, Ada 95 is an *integrated inhomogeneous* programming language. In Ada 95, active objects are called *tasks*. Tasks are syntactically described by a form very similar to the Ada *package*. A task has a *specification* describing the interface presented to other tasks (see section 10.3.4) and a *task body*. The task body may contain hidden data declarations, and contains a special sequence

of statements. Once a task is declared, the task is automatically *activated* and this sequence of statements is executed. A task terminates once it reaches its *end* statement.

A task declared in the declarative part of a subprogram, block or task body is said to depend on that unit. The rule is that a unit cannot be left until all dependent tasks have terminated. This termination rule ensures that objects declared in the unit and therefore potentially visible to dependent tasks cannot disappear while there still exists a task which could access them.

It is also possible to dynamically create tasks by declaring an access type to a task type. An instance of the task type can then be created by the evaluation of an allocator, just as objects are allocated on the heap. Tasks created in this way do not depend on the unit where they are created, but are dependent upon the block containing the access type declaration itself.

10.3.2 Task Identification

Access types to tasks can serve as a means for identifying tasks. But sometimes it is convenient to be able to refer to a task of any type. A server task might want to keep a record of past callers so that they can be recognized in the future.

This can be done using the package `Ada.Task_Identification`, which defines a type `Task_ID` and other operations upon tasks in general. The parameterless function `Current_Task` returns the identity of the currently executing task. The task identity of a task `T` is denoted by `T'Identity`.

10.3.3 Task Attributes

Not only is it useful to associate a unique identifier with a particular task, it can also be beneficial to assign other application dependent attributes to it. The *Systems Programming Annex* [ISO95, Annex C] offers the possibility to declare data structures for which there is a copy for each task in the system by means of the generic package `Ada.Task_Attributes`.

Instantiating the generic package somewhere in the program with an application specific data type results in creating an object of that type for each existing or subsequently created task in the system.

10.3.4 The Rendezvous

There are two main ways in which tasks can interact with each other in Ada: directly, by sending messages to each other, and indirectly, by accessing shared data.

Messages are directly passed between tasks in Ada by a mechanism known as the *rendezvous*. A rendezvous between tasks is similar to the human situation where two people meet, perform some action together, and then go on independently.

The rendezvous model of Ada is based on a client / server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). For that purpose, it declares one or more public *entries* in its task specification. Each entry identifies the name of the service, the parameters that are carried with the request and the results that will be returned.

A client task issues an *entry call* on the server task by identifying both the server and the requested entry. The server indicates its willingness to provide the service at any particular time by executing an *accept* statement as shown in figure 10.3.

```
task Server is
  entry Service (I : in Integer);
end Server;

task body Server is
  -- Declarations                          -- Client task
begin                                     declare
  ...                                       N : Integer := 0;
  accept Service (I : in Integer) do      begin
    -- Perform work                         Server.Service (N);
  end Service;                            end;
  ...
end;
```

Figure 10.3: The Rendezvous

For the communication to occur, *both* tasks must have issued their respective statements. When they have, the communication takes place. It is called a rendezvous because both tasks have to meet at the entry at the same time. Any `in` parameters are then passed from the client to the server, the server task executes the code inside the accept statement, any `out` parameters are passed back to the client, and only then both tasks proceed independently. It is of course possible that the client and the server will not both be in a position to communicate at exactly the same time. In this case, one task must wait for the other one. This synchronous form of communication is illustrated in figure 10.4.

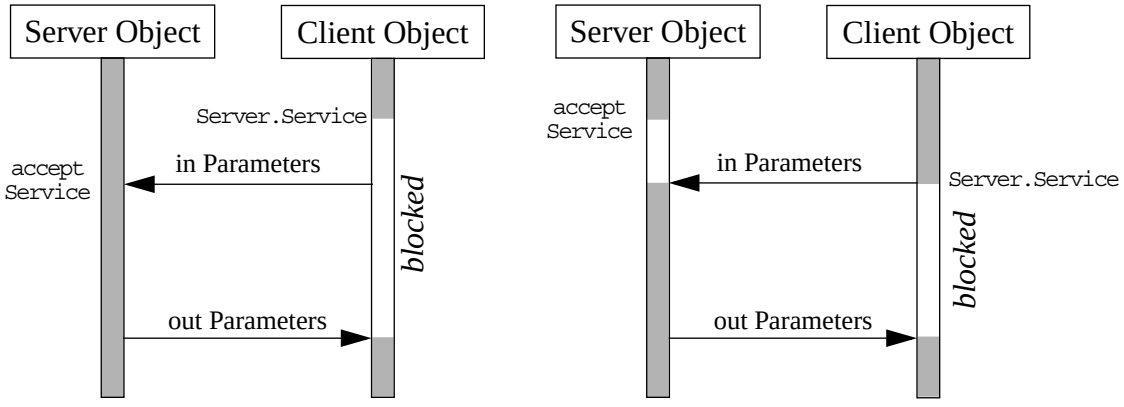

Figure 10.4: Synchronous Communication in the Rendezvous Model

Ada 95 allows more elaborate forms of rendezvous. By using the `select` construct, a server task can:

- Associate a condition with a rendezvous,
- Wait for more than a single rendezvous at any one time,
- Time-out if no rendezvous is forthcoming within a specified period,
- Withdraw its offer to communicate if no rendezvous is immediately available, or
- Terminate if no clients can possibly call its entries.

Likewise, a client task can also withdraw its offer to communicate if the server task does not immediately accept the call, or after some specified time-out.

10.3.5 Protected Types

One of the highlights of Ada has always been its integrated model for structuring concurrent computations using tasks and the rendezvous concept. The rendezvous model presents an abstraction from low–level synchronization primitives such as signals or semaphores.

However, experience with Ada over the years has shown that the rendezvous alone is not entirely sufficient to express synchronization in a convenient way. One problem is that rendezvous basically is control–flow oriented and does not lend itself easily to data synchronization. Shared data often had to be encapsulated within extra tasks in Ada 83, complicating the programs and leading to poor performance.

Ada 95 therefore introduced the concept of protected types. A protected type encapsulates some data items and offers synchronized access to this data through operations. Operations may be read–only (expressed as functions) or read–write (procedures and entries). The language guarantees mutual exclusion between all accesses to a protected object with the usual semantics of multiple readers or a single writer. In this respect a protected object is equivalent to a monitor [BH73]. A simple protected object encapsulating some data item that is to be accessed by several tasks might be written as shown in figure 10.5 below.

```
protected type Shared_Data is
  procedure Set (Val: in Data_Type);
  function  Get return Data_Type;
private
  Item : Data_Type;
end Shared_Data;
```

```
protected body Shared_Data is
  procedure Set
    (Val: in Data_Type) is
  begin
    Item := Val;
  end Set;
  function Get return Data_Type is
  begin
    return Item;
  end Get;
end Shared_Data;
```

Figure 10.5: A Protected Type for Mutual Exclusion

C. A. R. Hoare extended this basic monitor concept in [Hoa74] with so–called “condition variables” that could be declared inside a monitor. These basically are signals. A procedure of a monitor may suspend itself by waiting for the condition to become true. Another procedure of the monitor will signal on the condition variable to indicate that the condition has become true. In Hoare’s scheme, the `wait` operation relinquishes exclusion (to allow some other task to enter the monitor on the other procedure), and a `signal` operation immediately resumes a waiting task if there is one, making the signalling task leave the monitor.

The protected types of Ada 95 offer a similar feature, albeit in a more versatile way and at a higher level of abstraction. It has been noted that condition variables suffer from the same drawbacks as simple semaphores, notably that their correct use is not always easy and that the use of explicit `wait` and `signal` operations is just as error–prone as the use of `P()` and `V()` operations on a semaphore. Ada 95 solves this problem by introducing barriers: an `entry` of a protected type may have an associated *barrier*, a boolean expression that is (usually) expressed in terms of the state encapsulated in the protected type. A task calling an entry can only enter the protected object if the barrier is true; if not, the task is suspended until the barrier becomes true. Other tasks may enter the protected object through other entries or procedures, though. The run–time system checks at the end of each procedure or entry call whether any barriers on which tasks are waiting have become true, and if so, resumes a waiting task, letting it execute the entry. An example of a protected type implementing a bounded buffer is given in figure 10.6.

```
Max_Elems : constant Natural := ...;

type Count_Type is
  new Natural range 0 .. Max_Elems;

type Index_Type is
  new Natural range 1 .. Max_Elems;

type Buffer_Type is
  array (Index_Type) of Data_Type;

protected type Bounded_Buffer is

  entry Append (Val: in  Data_Type);

  entry Remove (Val: out Data_Type);

private

  Buffer       : Buffer_Type;
  First, Last  : Index_Type := 1;
  Size         : Count_Type := 0;

end Bounded_Buffer;
```

```
protected body Bounded_Buffer is

  entry Append (Val: in Data_Type)
    when Size < Max_Elems is
  begin
    Buffer (Last) := Val;
    Last := Last mod Max_Elems +1;
    Size := Size +1;
  end Append;

  entry Remove (Val: out Data_Type)
    when Size > 0 is
  begin
    Val := Buffer (First);
    First := First mod Max_Elems +1;
    Size := Size -1;
  end Remove;

end Bounded_Buffer;
```

Figure 10.6: A Protected Type with Entries

Ada 95 enhances this model of monitors even further by offering a few more language constructs:

- An attribute `'Count` may be applied within a protected object to one of its entries to obtain the number of tasks waiting on its barrier to become true. ("`Entry'Count > 0`" corresponds to Hoare's "`condition.queue`" predicate.)
- An attribute `'Caller` may be used within an entry to get the task identifier of the calling task.
- A `requeue` statement allows a programmer to requeue an entry call on the same or some other entry (in the latter case, the parameter profile must match).

These features greatly extend the expressive power of protected types. In particular, the `requeue` statement makes protected types capable of implementing preference control schemes at the application level; an example would be a resource allocation server that granted satisfiable requests but queued currently unsatisfiable requests for later servicing. In Ada 83, there was no satisfactory way to implement such servers.

10.3.6 Asynchronous Transfer of Control

Asynchronous transfer of control (ATC) is another important feature of Ada 95, allowing one task to signal another task without having to use a (synchronous) rendezvous. The language provides an asynchronous `select` statement of the form given in figure 10.7.

```
Asynchronous_Select :=
  select
    Triggering_Alternative
  then abort
    Abortable_Part
  end select;

Triggering_Alternative :=
  Triggering_Statement
  [Sequence_Of_Statements]
```

```
Abortable_Part[1] :=
  Sequence_Of_Statements

Triggering_Statement :=
  Entry_Call_Statement |
  Delay_Statement
```

1. Must not contain `accept` statements!

Figure 10.7: Syntax for Asynchronous `select` Statements

Using the asynchronous `select` statement, a task may alter its flow of control depending upon the asynchronous occurrence of external events. Applications of this include for instance mode changes in real–time systems, user interrupts or time–outs aborting lengthy computations, or error recovery [BW95].

The abortable part of an asynchronous `select` statement is started if the triggering statement is a delay that has not yet expired or an entry call that is queued (or later requeued using `requeue ... with abort`). If the triggering statement completes before the abortable part, the latter is aborted; otherwise, the triggering statement is aborted. The reader is referred to [ISO95, 9.7.4] for the complete rules (which are quite subtle).

If abortion were allowed to occur at any moment during the execution of the abortable part or of the trigger, it would be nearly impossible to maintain the consistency of the state accessed in an asynchronous select statement. The standard therefore defines a number of language constructs that defer abortions [ISO95, 9.8(6–11)], most notably protected actions and the `Initialize`, `Finalize`, and `Adjust` operations of controlled types. During these *abort–deferred* regions, abortions cannot occur: they are delayed until the abort–deferred region is left.

10.4 Integration of Concurrency and Object-Orientation in Ada

Arguably, Ada 95 does not fully integrate its models of concurrent and object-oriented programming [BW95]. For example, neither tasks nor protected objects are extensible. When Ada 95 was designed, the extensions to Ada 83 for object-oriented programming were, for the most part, considered separate to extensions to the concurrency model.

While working on my thesis, I participated in a research group that investigated ways of better integrating the concurrency model of Ada with its object-oriented programming model by allowing protected types to be extended, in a similar way as it is done for tagged types.

The requirements for extensible protected types are easy to articulate. In particular, extensible protected types should allow:

- new data fields to be added,
- new functions, procedures and entries to be added,
- functions, procedures and entries to be overridden,
- and class-wide programming to be performed.

These simple requirements raise many complex semantic issues.

10.4.1 Extensible Protected Types

This section presents the main ideas of extensible protected types by means of an example and mentions some of the problems associated with the approach. A complete analysis of potential solutions, corresponding problems (e.g. the so-called *inheritance anomaly* [MY93]), thoughts on implementation, and a concrete proposal for extending future versions of Ada can be found in [WJS+00a, WJS+00b].

In concurrent programming, *signals* are often used to inform tasks that events have occurred. Signals often have different forms: there are transient and persistent signals, there are those that wake up only a single task, and those that wake up all tasks. The following paragraphs illustrate how these abstractions can be built using extensible protected types.

Consider first an abstract definition of a signal as shown in figure 10.8.

```
package Signals is

  protected type Signal is
    abstract tagged (1)
    procedure Send;
    entry Wait is abstract;
  private
    Signal_Arrived : Boolean
      := True;
    entry Wait when Signal_Arrived;
  end Signal;   (2)

  type Signal_Ref is access
    all Signal'Class;

end Signals;
```

```
package body Signals is

  protected body Signal is
    abstract tagged

    procedure Send is (3)
    begin
      Signal_Arrived := True;
    end Send;

  end Signal;

end Signals;
```

Figure 10.8: Abstract definition of a Signal using Extensible Protected Types

For consistency with its usage in standard Ada, the word `tagged` indicates that a protected type is extensible (1). Just as in conventional object-oriented programming, the additional keyword `abstract` can be added, specifying that no instances of the type can be created. Normal protected types declare entry barriers inside the body of the protected type. To avoid having a child protected object depend on the body of its parent, it is necessary for extensible protected types to declare entry barriers in the private part of the specification (2). The body of the package contains the implementation of the procedure `Send` (3). There is no implementation for the abstract entry `Wait`.

Entries define the synchronization between operations on protected types. To avoid a major break of encapsulation, it is mandatory to have a way to reuse existing synchronization code defined for a parent class and to incrementally modify this inherited synchronization in a child class. In our proposal, entries in a derived protected type inherit the guards of the parent. In addition, a derived protected type is allowed to incrementally *strengthen* an inherited entry barrier by using the `and when` clause.

```
with Signals; use Signals;

package Persistent_Signals is

  protected type Persistent_Signal
    is new Signal with
    entry Wait;
  private
    entry Wait and when True;
  end Persistent_Signal;

end Persistent_Signals;
```

```
package body Persistent_Signals is

  protected body
    Persistent_Signal is

    entry Wait and when True is
    begin
      Signal_Arrived := False;
    end Wait;

  end Persistent_Signal;

end Persistent_Signals;
```

Figure 10.9: Deriving a Persistent Signal

A concrete signal, such as the `Persistent_Signal` that derives from the abstract `Signal` as shown in figure 10.9, must implement the entry `Wait`. In our example it is not necessary to

strengthen the guard of the entry. Syntactically, this is shown by adding the clause `and when True`.

Figure 10.10 shows how parts of the implementation of the parent type can be reused, just as often done in classic object-oriented programming. The protected type `Transient_Signal`, again derived from the abstract protected `Signal` type, overrides the procedure `Send`. `Send` checks the count of the entry `Wait` ①, and simply returns if there are no tasks waiting. Only if there are tasks queued on the entry, the signal is effectively sent by calling the `Send` procedure of the parent type ②.

```
with Signals; use Signals;
package Transient_Signals is
  protected type Transient_Signal
    is new Signal with
    procedure Send;
    entry Wait;
  private
    entry Wait and when True;
  end Transient_Signal;
end Transient_Signals;
package body Transient_Signals is
  protected body Transient_Signal is
    procedure Send is
    begin
      if Wait'Count = 0 then ①
        return;
      end if;
      Signal.Send; ②
    end Send;
    entry Wait and when True is ①
    begin
      Signal_Arrived := False;
    end Wait;
  end Transient_Signal;
end Transient_Signals;
```

Figure 10.10: Deriving a Transient Signal

Extensible protected types can also be used as generic parameters. Figure 10.11 illustrates this technique by defining a generic package that can be instantiated with any extensible protected type derived from the type `Signal`. The resulting package then provides a slightly modified version of the signal that, when a task calls `Send`, releases *all* tasks that are waiting on the entry `Wait` instead of just one.

```
generic
  type Base_Signal is new
    protected Signal;
package Release_All_Signals is
  protected type Release_All_Signal
    is new Base_Signal with
    entry Wait;
  private
    entry Wait and when True;
  end Release_All_Signal;
end Release_All_Signals;
package body Release_All_Signals is
  protected body Release_All_Signal
    is
    entry Wait and when True is
    begin
      if Wait'Count /= 0 then
        return;
      end if;
      Base_Signal.Wait;
    end Wait;
  end Release_All_Signal;
end Release_All_Signals;
```

Figure 10.11: A Generic Signal that Releases All Waiting Tasks

10.5 Distributed Systems in Ada

The previous section has detailed how Ada 95 supports lightweight concurrency by means of tasks. Heavyweight concurrency is provided in Ada 95 through the notion of *partitions*. Ada 95 is the first programming language that defines a precise model for the development and structuring of distributed applications in the language standard [ISO95, Annex E].

In an abstract way, a distributed application can be viewed as a collection of cooperating entities or program fragments (so–called "virtual nodes") that themselves are indivisible as far as distribution is concerned and that are distributed on various computers (or "physical nodes") in a network, over which they communicate with each other.

A virtual node in Ada 95 is called a *partition*[1]. A partition is a collection of library units, some of which constitute its interface towards the other partitions of the application. These well–defined interfaces are given by the specifications of certain library units that have to be marked in the source using special categorization pragmas as belonging to the interface.

Ada 95 distinguishes *active* and *passive* partitions. Passive partitions are intended to model memory shared between virtual nodes, either physically or through a virtual distributed shared memory system, and have no thread of control associated with them. Their interface may contain remotely accessible data objects, which can be accessed by other (active) partitions. Active partitions *do* have a thread of control. Different active partitions communicate with each other only through their interface library units by means of remote procedure calls. Direct remote data access between active partitions does not exist in Ada 95.

Partitions are semi–autonomous: while they only make sense as part of a larger distributed application and thus are intrinsically bound to cooperate with the other partitions, they evolve independently from each other with regard to tasking, time, I/O, and so on. The language standard does not require a distributed run–time support that might offer a common base for such services. A direct consequence of this is that all tasks are local to a partition; they're not visible across partitions and hence there is no remote rendezvous in distributed Ada 95.

The configuration of distributed applications, which covers the assignment of library units to partitions and the allocation of partitions to physical nodes, is beyond the language standard. Extensions of the standard are allowed, and even encouraged. [KWS96, PW97] present transparent encryption of data sent over the network, [Kie97] investigates support for dynamic client-server applications, and [WS99] addresses transparent replication for Ada partitions.

1. Not to be confused with a *network partition*, i.e. the rupture of communication links such that a formerly connected network is split in several parts that cannot communicate with each other anymore.

10.5.1 Remote Procedure Calls

Communication between active partitions is based solely on the concept of remote procedure calls [BN84], RPC for short, a communication abstraction for interpartition communication built on top of simple message passing. A remote procedure call in Ada 95 is transparent to the application developer: both the implementation of a remotely callable subprogram as well as its call do not differ at all from a local subroutine.

The internal working of a remote procedure call can be decomposed into five distinct phases, as shown in figure 10.12.

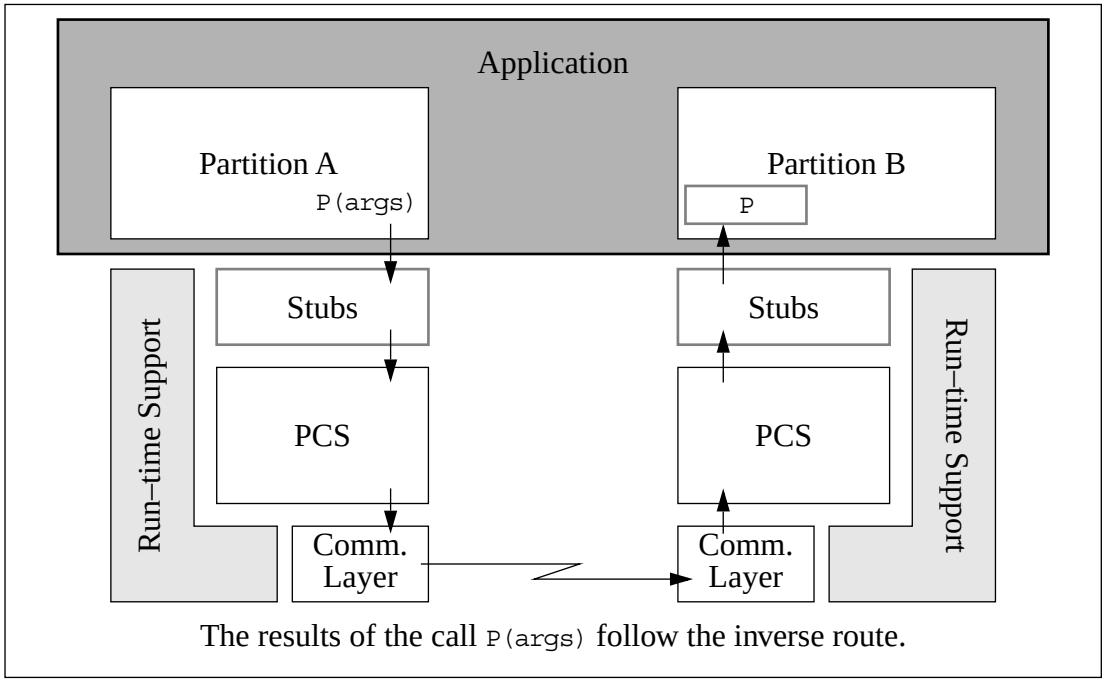

Figure 10.12: Schematic View of a Remote Procedure Call

1. In order to send the arguments of the call over the network, they must be flattened into a single stream of bytes. This process is called *marshaling* and is performed transparently for the application in the *caller's stub* version of the remotely callable subprogram. The caller's stub also adds some identification of the routine to be invoked to this stream of bytes. The run–time support then sends an *RPC request message* to the partition where the implementation of the desired subprogram resides.
2. On the receiving side, the run–time support gets the information which subprogram is to be called from the request message and invokes the corresponding *receiver's stub* procedure. The receiver's stub then reconstructs an internal representation of the arguments from the byte stream in the request message (this is called *unmarshaling*) and locally invokes the correct subprogram.
3. The remotely callable subprogram executes.

4. The receiver's stub marshals all the values returned from the subprogram, i.e. `inout`– and `out`–parameters, function results, or even the identity of an exception, if one was raised in step 3. The run–time support then sends an *RPC answer message* back to the calling partition, containing the results of the call (if any).
5. On the calling partition, the run–time support passes this answer message back to the caller's stub, which then unmarshals the results and passes them back (possibly including the raising of an exception) to the original point of call in the application.

The previous description is that of the basic remote procedure call in Ada 95. It is called a *synchronous* or *blocking* RPC, because the caller proceeds only after having received the answer message. The language standard also offers so–called *asynchronous* or *non–blocking* RPCs, in which the steps 4 and 5 of the previous sequence are omitted. The caller can continue its execution after having sent the request message, possibly before the remotely called subprogram has finished its execution. Asynchronous RPCs are a way to provide a type–safe interface to message passing in Ada 95. Asynchronously called operations cannot return results and may have only parameters of mode `in`. Any exceptions they might raise are lost on the caller's side; they are not propagated to the calling partition. Whether a certain remotely callable subprogram is to be called asynchronously is defined statically by applying pragma `Asynchronous` to the declaration of the subprogram.

The callers' and receivers' stubs are generated automatically by the compiler or by a partitioning tool. The marshaling and unmarshaling process uses the standard *Stream* facility of Ada 95 presented in section 10.7 on page 148. An application having special needs regarding the conversion of certain types into byte streams may provide its own implementations for marshaling and unmarshaling. The message passing protocol underlying a remote procedure call and described previously is completely hidden from the application in the *Partition Communication Subsystem*, PCS for short, which is part of the run–time support. The PCS is invoked by the stubs through a standardized interface in package `System.RPC`.

10.5.2 Distributed Objects

The distribution model of Ada 95 also covers the realm of object–oriented programming. The language standard uses both the new object-oriented features and the new distribution features to define a distributed object model, which adds great flexibility to both areas.

An object in Ada 95 always resides on the partition it was created on; despite the designation "distributed object", it cannot migrate from one partition to another one. However, it *is* possible to have remote accesses to objects that reside on other partitions. In conjunction with dispatching calls, such remote references provide *remote dispatching* as a further means to dynamically designate the destination of a remote call.

It is possible to have objects of the same derivation class or even of the same tagged type on two different partitions. A third partition may obtain remote references to such

objects in the form of values of a remote access–to–class–wide type. This third partition can then use dispatching calls to invoke primitive operations on the partition that created the object without having to know explicitly *which* partition that is. The receiving partition is transparently embedded in the remote access value.

10.5.3 Fault Tolerance in Distributed Ada

The language standard for Ada 95 does not require distributed applications to be fault–tolerant at all. The only provision made is the predefined exception `Communication_Error` that may be raised on the partition initiating a remote call when the destination partition is inaccessible. Apart from that, [ISO95, E.1(12)] specifically states that an implementation may allow replication of partitions, but this is only an implementation permission, not a requirement.

10.6 Exceptions in Ada

Ada 83 was one of the first mainstream programming languages incorporating exceptions. Exceptions in Ada are provided to address the following situations:

- *Error conditions* — like arithmetic overflow, storage exhaustion, array-bound violations, subrange violations, peripheral time-outs, etc. When one of these situations arises, the Ada run-time notifies the application programmer by means of predefined exceptions, e.g. `Constraint_Error`, `Storage_Error`, or exceptions defined in predefined library units, e.g. `Communication_Error`.
- *Abnormal program conditions* — like errors in user input data, need for special algorithms to deal with singularities, or incorrect usage of abstract data types, etc. To address these situations, Ada allows the application programmer to define new exceptions. Later on, when the abnormal condition occurs, the programmer may raise an exception explicitly.

The latter is illustrated in figure 10.13. The generic package `Stacks` implements a bounded stack as an abstract data type. The upper limit of elements that the stack can hold is chosen at package instantiation time. When this maximum number has been reached, invoking the `Push` operation leads to a situation in which the operation can not be completed in the usual way. In that case, the exception `Stack_Overflow` is raised.

When an exception has occurred, the control is passed to a specified sequence of statements which is called the *exception handler.* Exception handlers are separated, textually, from the place at which the error is raised so that the normal behavior of the program is not obscured. The Ada model of exceptions is based on the termination model [Goo75], and

```
generic
  type Element is private;
  Max_Elements : Natural := 100;
package Stacks is

  type Stack is private;

  procedure Push ( S : in out Stack;
                   E : in Element);

  Stack_Overflow : exception;

  ...

private

  type Stack is ...
  --  Bounded stack implementation

end Stacks;
```

```
package body Stacks is

  procedure Push ( S : in out Stack;
                   E : in Element) is
  begin
    ...
    if Condition then
      raise Stack_Overflow;
    end if;
    ...
  end Push;

  ...

end Stacks;
```

Figure 10.13: Exception Declaration and Explicit Raising

does therefore not allow for an automatic return to the point of the error from within the exception handler.

Any program block or subprogram may contain handlers that can catch exceptions raised during the execution of that block. The `Stacks` package declares the exception `Stack_Overflow` in its interface. This allows users of stacks to catch the exception and take appropriate measures, as shown in figure 10.14.

Unfortunately, Ada does not allow to associate exceptions with subprograms, only with an entire package. This is rather surprising, given that the Ada programming language tries to reduce programming errors by enforcing strict typing rules, by providing safe language constructs, and by performing run-time checks. Java for instance allows to associate exceptions with methods. As a result, the compiler can verify that the exceptions declared in a method's interface are handled in the code that calls the method. Otherwise, the exceptions must also be part of the interface of the calling method.

```
declare

  package My_Stacks is
    new Stacks (Integer, 10);
  use My_Stacks;

  S : My_Stacks.Stack;

begin

  ...
  My_Stacks.Push (123);

exception

  when Stack_Overflow =>
    -- Handle error

end;
```

Figure 10.14: Exception Handling

In Ada, unhandled exceptions in a block are propagated to the calling block. This fact, and the fact that exception names can be declared in any declarative region, makes it possible for exceptions to be propagated outside of the scope of the exception name, turning them into so-called *anonymous exceptions*. To still catch these exceptions, the optional handler "others" is provided. It guarantees to catch all exceptions raised in the block, excluding those already mentioned and therefore explicitly handled.

10.6.1 The Package Ada.Exceptions

Exceptions in Ada are not objects, and therefore it is not possible to create exception hierarchies or attach data to exceptions by declaring new attributes.

Fortunately, the package Ada.Exceptions provides further facilities for manipulating exceptions. It introduces the concept of an exception occurrence, and a number of subprograms to access information about the occurrence. Using these subprograms it is for instance possible to obtain a string representation of the exception name, even for anonymous exceptions, or to get information in form of a printable string describing the details of the cause of an exception occurrence.

The package also provides a subprogram named Raise_Exception that allows the programmer to attach a specific message in form of a character string to an exception when raising it. During exception handling, this message can be retrieved by calling the Exception_Message function. It is also possible to save or copy an exception occurrence, and then raise it again later on.

10.7 Persistence in Ada

Although Ada 95 does not provide direct support for data persistence, it nevertheless introduced a new concept called streams that greatly facilitates implementing persistence.

A stream is a sequence of elements comprising values from possibly different types. The values stored in a stream can only be accessed sequentially. Ada streams can be seen as one of the first incarnations of the *Serializer* design pattern described in section 5.2.3 on page 70.

The standard package Ada.Streams defines the interface for streams in Ada 95 [ISO95, 13.13.1]. Its specification is shown in figure 10.15.

The Ada.Streams package declares an abstract type Root_Stream_Type, from which all other stream types must derive. Every concrete stream type must override the Read and Write operations, and may optionally define additional primitive subprograms according to the functionality of the particular stream. Obviously, the root stream type plays the *Reader* / *Writer* role in the *Serializer* pattern. Derivations of the root stream type incarnate the *ConcreteReader* / *ConcreteWriter* and the backend interface.

In Ada 95, the predefined attributes 'Write and 'Output are used to write values to a stream, thus converting them into a flat sequence of stream elements. Reconstructing the values from a stream is done with the predefined attributes 'Read and 'Input. They make dispatching calls on the Read and Write procedures of the Root_Stream_Type. When using 'Write and 'Read, neither array bounds nor tags of tagged types are written to or read from the stream. 'Output and 'Input are used for that purpose.

```
package Ada.Streams is

   type Root_Stream_Type is abstract tagged limited private;

   type Stream_Element is mod implementation-defined;

   type Stream_Element_Offset is range implementation-defined;

   subtype Stream_Element_Count is Stream_Element_Offset
      range 0..Stream_Element_Offset'Last;

   type Stream_Element_Array is
      array (Stream_Element_Offset range <>) of Stream_Element;

   procedure Read (Stream : in out Root_Stream_Type;
                   Item   : out Stream_Element_Array;
                   Last   : out Stream_Element_Offset) is abstract;

   procedure Write (Stream : in out Root_Stream_Type;
                    Item   : in Stream_Element_Array) is abstract;

private
   ... -- not specified by the language
end Ada.Streams;
```

Figure 10.15: The Package `Ada.Streams`

All non-limited types have default implementations of the stream attributes, hence all non-limited types implement the *Serializable* interface and can therefore serve as *ConcreteElement*. It is possible to replace the default implementation of the stream attributes for any type via an attribute definition clause. In order to write a value of a limited type to a stream, such an attribute definition clause is even mandatory. Any procedure having one of the predefined signatures shown in [ISO95, 13.13.2] can replace the default implementation. Figure 10.16 shows how to replace the predefined implementation of `'Write` for an integer type.

```
type My_Integer is new Integer;

procedure My_Write (Stream : access Ada.Streams.Root_Stream_Type'Class;
                    Item   : My_Integer);

for My_Integer'Write use My_Write;
```

Figure 10.16: Overriding the Default `Write` Procedure

The only concrete stream implementation that is defined in the language standard is `Stream_IO` [ISO95, A.12], a child package of `Ada.Streams`. It provides stream-based access to files. `Stream_IO` offers also file manipulation operations such as `Create`, `Open`, `Close`, `Delete`, etc. Figure 10.17 shows how to write values of elementary types, array types and tagged types to a file by means of a stream and how to reconstruct them again.

```
with Ada.Streams.Stream_IO;
use Ada.Streams.Stream_IO;

--  writing
declare

  My_File: File_Type;
  S : Stream_Access;
  I : Integer;
  My_String : String := "Hello";
  T : A_Tagged_Type'Class := … ;

begin

  Create (My_File "file_name");
  S := Stream (My_File);

  --  do some work
  Integer'Output (S, I);
  String'Output (S, My_String);
  A_Tagged_Type'Class'Output (S, T);
  Close (My_File);

end;
```

```
--  reading
declare
  My_File : File_Type;
  S : Stream_Access;
begin

  Open (My_File, "file_name");
  S := Stream (My_File);
  declare
    I : Integer :=
      Integer'Input (S);
    My_String: String :=
      String'Input (S);
    T : A_Tagged_Type :=
      A_Tagged_Type'Class'Input (S);
  begin
    --  do some work
  end;

  Close (My_File);

end;
```

Figure 10.17: Writing and Reading Data to / from a Stream

Chapter 11:
Implementation for Ada 95

In the previous chapter we have seen that Ada 95 provides good support for object-orientation, basic exception handling features, support for streaming and elaborate mechanisms for dealing with concurrency. This chapter presents how these features have been used to implement the OPTIMA framework and to provide concrete implementations of the interfaces presented in chapter 9 for the application programmer.

11.1 Implementing the Framework

11.1.1 Objects

In general, all classes of the framework have been implemented as Ada tagged types, encapsulated inside an Ada package. Subclasses have been implemented as extensions of the tagged type declared in child packages as illustrated in section 10.2 on page 132.

Of course, special attention must be paid when implementing a class that might be accessed concurrently. The state of such a class must be encapsulated inside a protected object in order to preserve data consistency. Since Ada does not provide inheritance for protected types, the protected type must be made a component of a tagged type. This is, for instance, the case for the `Transaction` class, because it handles concurrent calls to `Join_Transaction`, `Close_Transaction`, `Commit_Transaction` and `Abort_Transaction`.

Ada 95 does not provide automatic garbage collection. In order to prevent memory shortage, it is important to destroy all objects once they are not needed anymore. This is not trivial, especially for objects that are shared among multiple tasks. The problem can be

solved by keeping track of all references that point to a given object. Once the last reference disappears, the object can be safely destroyed.

Reference counting can be achieved in Ada by using controlled types as shown in figure 11.1. The `Controlled_Object_Ref` type is a controlled type that encapsulates a reference to the actual object ①. When invoking the constructor function `Create_Object`, the actual object is created and an instance of `Controlled_Object_Ref` is handed back to the caller instead of an ordinary reference ②. The actual object contains a counter that is initialized to one at creation time. Each time the controlled reference is duplicated, the `Adjust` procedure increments that counter ③. If a controlled reference goes out of scope, the `Finalize` procedure decrements the counter ④. If the counter reaches zero, the system knows for sure that nobody is using the object anymore and can safely destroy the object and reclaim the associated memory.

```
package Reference_Counting is

  type Controlled_Object_Ref is
    private;

  function Create_Object (...)
    return Controlled_Object_Ref;

  --  Other operations

private

  type Controlled_Object_Ref is new
    Ada.Finalization.Controlled with
    record
    Tracked_Object : Object_Ref; ①
  end record;

  procedure Adjust (Ref : in out
    Controlled_Object_Ref);

  procedure Finalize (Ref : in out
    Controlled_Object_Ref);

end Reference_Counting;

package body Reference_Counting is

  function Create_Object (...)
    return Controlled_Object_Ref is
    Result : Controlled_Object_Ref;
  begin
    Result.Tracked_Object :=
      new Object (...);
    Set_Reference_Count>
      (Result.Tracked_Object, 1);
    return Result; ②
  end Create_Object;

  procedure Adjust (Ref : in out
    Controlled_Object_Ref) is
  begin
    if Ref.Tracked_Object /= null
      then
      Increase_Reference_Count
        (Ref.Tracked_Object); ③
    end if;
  end Adjust;

  procedure Finalize (Ref : in out
    Controlled_Object_Ref) is
    if Ref.Tracked_Object /= null
      then
      Decrease_Reference_Count
        (Ref.Tracked_Object); ④
    end if;
  end Finalize;

end Reference_Counting;
```

Figure 11.1: Reference Counting with Controlled Types

This technique is for instance used to determine when a `Transaction` object (see section 6.5 on page 77) is not used anymore. Transaction objects are accessed through `Transaction_Identifier` objects, which implement reference counting as described above.

11.1.2 Concurrency Control

Some of the advanced concurrency features of Ada 95 have been used inside the implementation of the concurrency control components. The example shown in this section is taken from the implementation of the lock manager, which handles cooperative and competitive concurrency control for pessimistic concurrency control schemes based on locking.

The lock manager must handle multiple tasks, and therefore, its functionality has been implemented inside a protected type as shown in figure 11.2. The lock manager must implement four operations, namely `Pre_Operation`, `Post_Operation`, `Commit_Transaction` and `Abort_Transaction` (see section 7.3 on page 82). They map to the four operations in the public part of the `Lock_Manager_Type` specification. Each time a participant of a transaction wants to execute an operation on a transactional object, the `Pre_Operation` operation of the associated lock manager is invoked. The lock manager verifies that the participant can safely access the object. If this is not the case, then the calling task is suspended. Therefore, `Pre_Operation` is a potentially blocking operation, and must be implemented as an entry.

The private part of the specification contains three private entries and some attributes, e.g. the set of granted operations named `My_Info`, and a boolean and a natural variable that are used to implement multiple readers / single writers.

```
protected type Lock_Manager_Type is

   entry Pre_Operation (Info : Operation_Information_Ref;
                        Transaction : Transaction_Ref);

   procedure Post_Operation;

   procedure Transaction_Commit (Transaction : Transaction_Ref);

   procedure Transaction_Abort (Transaction : Transaction_Ref);

private

   entry Waiting_For_Transaction (Info : Operation_Information_Ref;
                                  Transaction : Transaction_Ref);

   entry Waiting_For_Writer (Info : Operation_Information_Ref;
                             Transaction : Transaction_Ref);

   entry Waiting_For_Readers (Info : Operation_Information_Ref;
                              Transaction : Transaction_Ref);

   Currently_Writing : Boolean := False;
   Currently_Reading : Natural := 0;
   To_Try : Natural := 0;
   My_Info : Info_Set_Type;

end Lock_Manager_Type;
```

Figure 11.2: The `Lock_Manager` Specification

The implementation of the `Pre_Operation` operation is shown in figure 11.3. It illustrates the two phases of concurrency control that must be applied to transactional objects. First, competitive concurrency must be handled. To guarantee the serialization property of trans-

actions, the lock manager must determine if the operation to be invoked does not conflict with other operation invocations by still active transactions. This check is performed in the `Is_Compatible` function ①. If a conflict has been detected, the calling task is suspended by requeuing the call on the private entry `Waiting_For_Transaction` until the transaction having executed the conflicting operation ends ②.

If, on the other hand, the first phase completes successfully, then the operation information is inserted into the list of granted operations ③ and the second phase is initiated by requeuing on the private entry `Waiting_For_Writer` ④.

```
entry Pre_Operation (Info : Operation_Information_Ref;
                     Transaction  : Transaction_Ref) when True is
begin
  if not Is_Compatible (My_Info, Info, Transaction) ① then
    requeue Waiting_For_Transaction with abort; ②
  else
    Insert (My_Info, Info, Transaction); ③
    requeue Waiting_For_Writer with abort; ④
  end if;
end Pre_Operation;

entry Waiting_For_Writer (Info : Operation_Information_Ref;
                          Transaction : Transaction_Ref)
  when not Currently_Writing and Waiting_For_Readers'Count = 0 ⑦ is
begin
  if Is_Modifier (Info.all) then ⑤
    if Currently_Reading > 0 then
      requeue Waiting_For_Readers with abort; ⑥
    else
      Currently_Writing := True;
    end if;
  else
    Currently_Reading := Currently_Reading + 1;
  end if;
end Waiting_For_Writer;

entry Waiting_For_Readers (Info : Operation_Information_Ref;
                           Transaction : Transaction_Ref)
  when Currently_Reading = 0 is
begin
  Currently_Writing := True;
end Waiting_For_Readers;

procedure Post_Operation is
begin
  if Currently_Writing then
    Currently_Writing := False;
  else
    Currently_Reading := Currently_Reading - 1;
  end if;
end Post_Operation;
```

Figure 11.3: Implementing Cooperative Concurrency Control

The two entries `Waiting_For_Writer` and `Waiting_For_Readers` implement the multiple readers / single writer paradigm. Starvation of writers is prevented by keeping readers and writers on a single entry queue. Requests are serviced in FIFO order.

By calling `Is_Modifier` ⑤, the lock manager determines the nature of the operation, i.e. read or write. Read operations are allowed to proceed, until a write operation is encountered. If there are still readers using the transactional object, then the writer is requeued to the `Waiting_For_Readers` entry ⑥. This closes the barrier of the `Waiting_For_Writer` entry, since the latter requires the `Waiting_For_Readers` queue to be empty ⑦.

After the invocation of the actual operation on the transactional object, `Post_Operation` is called, which decrements the number of readers respectively unsets the writer flag.

Now let us go back to the first phase and see what happens to the calls queued on the `Waiting_For_Transaction` entry. Tasks queued here have requested to execute an operation that conflicts with an operation already executed on behalf of a still active transaction. Each time a transaction ends, this situation might change. When a transaction aborts, the operations executed on behalf of the transaction are undone, and hence the locks can be released. This is illustrated in Figure 11.4. `Transaction_Abort` calls the `Delete` operation of the granted operation set ⑴ₐ, which results in removing all operation information of the corresponding transaction from the set. The same is done upon commit of a top-level transaction ⑴ᵦ. If the commit involves a subtransaction, then the locks held so far by the subtransaction must be passed to the parent transaction. This is achieved by calling `Pass_Up` ②.

```
entry Waiting_For_Transaction (Info : Operation_Information_Ref;
                               Transaction : Transaction_Ref) when To_Try > 0 is
begin
  To_Try := To_Try - 1;
  requeue Pre_Operation with abort;
end Waiting_For_Transaction;

procedure Transaction_Commit (Transaction : Transaction_Ref) is
begin
  if Is_Toplevel (Transaction.all) then
    Delete (My_Info, Transaction); (1b)
  else
    Pass_Up (My_Info, Transaction, Get_Parent (Transaction.all)); (2)
  end if;
  To_Try := Waiting_For_Lock'Count; (3)
end Transaction_Commit;

procedure Transaction_Abort (Transaction : Transaction_Ref) is
begin
  Delete (My_Info, Transaction); (1a)
  To_Try := Waiting_For_Lock'Count; (3)
end Transaction_Abort;
```

Figure 11.4: Implementing Competitive Concurrency Control

In any case, the auxiliary variable `To_Try` is set to the number of tasks waiting in the queue of the entry `Waiting_For_Transaction` ③. As a result, all queued tasks are released and requeued to `Pre_Operation`, thus getting another chance to access the transactional object.

11.1.3 Persistence

The implementation of the persistence support [KR00] is based on Ada streams (see section 10.7 on page 148), which allow to transform any non-limited object into a flat stream of bytes. The stream concept has been extended to support buffering and different kinds of storage devices as presented in section 8.2.1 on page 91. This fits exactly the idea behind Ada streams, which is that programmers can develop their own stream subclasses by inheriting from the given abstract class.

11.1.3.1 The Storage Hierarchy

Figure 11.5 shows the specification of the `Storage_Type` class, which implements the root class of the storage hierarchy presented in figure 8.4 on page 94.

```
with Ada.Streams; use Ada.Streams;
with Ada.Finalization; use Ada.Finalization;

package Storage_Types is

   type Storage_Type (<>) is abstract tagged limited private;

   type Storage_Ref is access all Storage_Type'Class;

   procedure Read (Storage : in out Storage_Type;
                   Item    : out Stream_Element_Array;
                   Last    : out Stream_Element_Offset) is abstract;

   procedure Write (Storage : in out Storage_Type;
                    Item    : in Stream_Element_Array) is abstract;

private

   type Storage_Type is new Limited_Controlled with null record;

end Storage_Types;
```

Figure 11.5: Specification of type `Storage_Type`

`Storage_Type` is privately derived from `Limited_Controlled` in order to allow concrete storage implementations to perform automatic initialization and finalization, if necessary. Disk files for instance should always be closed, network ports should be freed, etc. `Storage_Type` is limited, so it can store, if necessary, other limited data, such as for example file descriptors. Finally, the public view of `Storage_Type` has unknown discriminants. As a consequence, the user of a storage type is forced to call one of the constructor functions of a concrete storage type; he or she can not just declare an instance of the type and thereby bypass correct initialization.

The operations provided by `Storage_Type` are `Read` and `Write`. They have the same signature as the ones defined for `Root_Stream_Type` (see section 10.7 on page 148).

Whenever streams are used to access storage devices, it is not always a good idea to write the data to the device on every call to `'Write` or `'Output`. At what time the data should be written to the device is largely device dependent. Disk devices for example are usually accessed in fixed-sized chunks of data called blocks. In this case, too many individual write accesses can result in considerable performance loss. It is much more efficient to buffer the data.

In some situations, buffering is even mandatory. The transaction framework, for example, uses stable storage to store the transaction log. The recovery algorithm depends on the fact that updates to the log are performed in an atomic manner, i.e. are either done completely, or not done at all. The storage hierarchy requires all stable storage to implement atomic writes, but this atomicity can be broken if an object is written to an unbuffered stream. As explained in “The Serializer Design Pattern” on page 70, there is often a recursive back-and-forth interplay between the *Reader* / *Writer* and the object implementing the *Serializable* interface. Likewise, the `'Read` and `'Write` attributes may make multiple calls to the `Read` and `Write` procedures of a stream when serializing the state of an object. To still achieve atomic writing, buffering is necessary.

11.1.3.1 The Buffer Hierarchy

Buffers come in two flavors, *bounded* and *unbounded* as shown in figure 11.6.

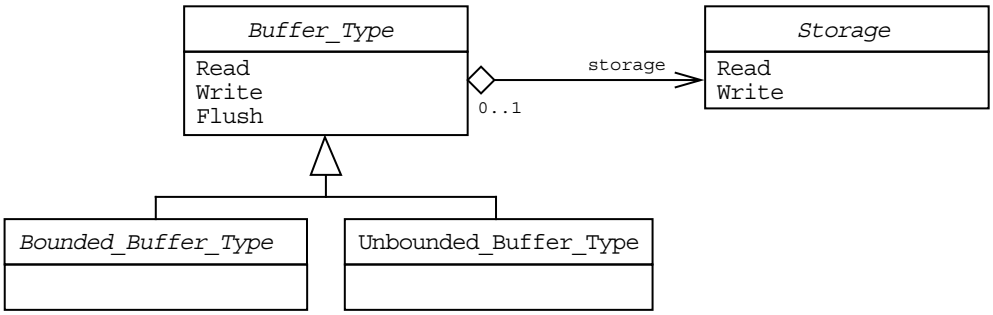

Figure 11.6: The *Buffer* Hierarchy

Like storage, buffers provide the `Read` and `Write` operations, and an additional operation `Flush`. Inside the `Write` procedure, the data is first written into a memory buffer, and only when `Flush` is called, the data is written out to the corresponding storage. The `Read` operation does the inverse: it will try and read all the data or as much data as possible from the storage device into the buffer. Subsequent calls can then be served without accessing the storage.

When implementing the unbounded buffer class, it is possible to use an instance of the volatile memory storage type to buffer the data. This illustrates the possibilities of reuse offered by the storage hierarchy.

11.1.3.1 Normal and Buffered Streams

Finally, the package `Streams` provides the link between Ada streams, buffers and the storage hierarchy. It defines two concrete stream types, `Stream_Type` and `Buffered_Stream_Type`, both descendants of `Abstract_Stream_Type`, which in turn is derived from `Ada.Streams.Root_Stream_Type`. The specification of the package is presented in figure 11.7.

```
with Ada.Streams; use Ada.Streams;
with Buffer_Types; use Buffer_Types;

package Streams is

  type Abstract_Stream_Type is abstract new
    Ada.Streams.Root_Stream_Type with private;

  type Abstract_Stream_Ref is access all Abstract_Stream_Type'Class;

  type Stream_Type (Storage : access Storage_Type'Class)
    is new Abstract_Stream_Type with private;

  procedure Read (Stream : in out Stream_Type;
                  Item   : out Stream_Element_Array;
                  Last   : out Stream_Element_Offset);

  procedure Write (Stream : in out Stream_Type;
                   Item   : in Stream_Element_Array);

  type Buffered_Stream_Type (Buffer : access Buffer_Type'Class)
    is new Abstract_Stream_Type with private;

  procedure Read(Stream: in out Buffered_Stream_Type;
              Item : out Stream_Element_Array;
              Last : out Stream_Element_Offset);

  procedure Write(Stream: in out Buffered_Stream_Type;
              Item : in Stream_Element_Array);

  procedure Flush(Stream: in out Buffered_Stream_Type);

end Streams;
```

Figure 11.7: Specification of the `Streams` package

The package allows the user to choose between a normal stream (one that writes the data to the storage medium on every `'Write`) and a buffered stream (one that buffers the data until the user calls `Flush`). The type of buffer and the type of storage that will be used for the stream must be chosen at object creation time through an access discriminant, following the ideas of the *Strategy* design pattern (see section 5.2.2 on page 69).

11.2 Transaction Framework Interfaces for Ada 95

The rest of this chapter covers the interface between the Ada programmer and the transaction framework.

11.2.1 Transaction Identifier Management

Once a task is part of a transaction, it can invoke operations on transactional objects. The recovery and lock manager must know on behalf of which transaction the operation is executed. Most systems therefore need to pass a transaction identifier as a parameter to every operation of a transactional object.

The *Systems Programming Annex* [ISO95, Annex C] of the Ada Standard offers the possibility to declare data structures for which there is a copy attached to each task in the system by means of the generic package `Ada.Task_Attributes`. This technique, illustrated in figure 11.8, has been used to associate each task running on behalf of a transaction with the corresponding transaction object. When calling an operation on a transactional object, the run-time support can retrieve the transaction identifier using the function `Task_Attributes.Value`. This means in practice that, for the application programmer, there is no difference between calling a transactional object and calling a normal object.

```
with Transaction_Identifiers; use Transaction_Identifiers;

type Transactional_Task_Attribute_Type is
  new Ada.Finalization.Controlled with record
    Current_Tid : Transaction_Identifier_Ref := null;
end record;

procedure Finalize (Attribute : in out Transactional_Task_Attribute_Type);

Null_Task_Attribute : constant Transactional_Task_Attribute_Type
  := (Ada.Finalization.Controlled with Current_Tid => null);

package Transactional_Tasks is new
  Ada.Task_Attributes (Transactional_Task_Attribute_Type,
                       Null_Task_Attribute);
```

Figure 11.8: Associating the Transaction Context with a Task

By making the task attribute *controlled*, it is possible to execute a set of statements on every task termination. The `Finalize` procedure of the controlled type is automatically invoked before the task is destroyed. As a result, it is possible to detect *deserter* tasks, i.e. tasks that disappear without voting on the outcome of a transaction. The open multithreaded transaction model treats deserter tasks as an error (see section 4.6.3 on page 59). Therefore, if a terminating task is still associated with a transaction, the `Finalize` procedure contacts the transaction support component and aborts the transaction.

11.2.2 Encapsulating Data Objects

As presented in section 9.2 on page 110, any object used in a transaction must be transformed into a transactional object by programming a wrapper object that, for each operation invocation, performs the necessary calls to the transaction framework.

Programming such a wrapper object is a mechanical process, and could be done by a preprocessor. In Ada, such a "transactional object generator" could be implemented by using the *Ada Semantic Interface Specification* [ISO99], ASIS for short. ASIS is an interface between an Ada Environment and any tool or application requiring statically-determinable information about an Ada program.

Of course, if the programmer wants to use semantic-based concurrency control for controlling access to some transactional object, then the generation of the needed operation information classes (see section 7.4.3 on page 87) can not be automated completely, since only the programmer can reliably determine the semantics of the operations of the data object. In this case, a semi-automatic approach could be developed.

Up to now, such a preprocessor has not been implemented, and therefore the programmer must hand-code the transactional object wrapper for each type of data object. Section 13.3.1 on page 198 shows an example of how to construct a transactional wrapper object for an account data type.

11.2.2.1 Interfacing with the Cache Manager

Every transactional object has an associated memory object, which encapsulates the original data object. When creating or restoring a transactional object, such a memory object must be obtained from the cache manager. For this purpose, the transaction interface offers the two procedures presented in figure 11.9.

The first parameter of the `Create` function identifies the storage unit on which the data object is to be created. The other parameters are concrete implementations of the classes presented in figure 9.2 on page 113. They allow the memory object to create, load, save and delete objects of the data type that is to be encapsulated. The `Logging` and `Update` parameter determine the kind of logging technique and the kind of update strategy that are to be used. If the storage unit is already in use, then the exception `Object_Exists` is raised. The `Storage_Device_Error` exception is raised in case some storage related error arises.

The `Restore` function tries to initialize the state of the data object with data stored on the storage unit. If the storage unit can not be found, the exception `Object_Non_Existant` is raised. The `Object_Corrupt` exception is raised in case the data on the storage unit has been corrupted.

```
type Logging_Kind is (Physical, Logical);

type Update_Kind is (Inplace, Deferred);

  function Create (Params   : Non_Volatile_Params_Type'Class;
                   Creation : Creation_Operation_Type'Class;
                   Loading  : Loading_Operation_Type'Class;
                   Saving   : Saving_Operation_Type'Class;
                   Deletion : Deletion_Operation_Type'Class;
                   Logging  : Logging_Kind;
                   Update   : Update_Kind)
    return Memory_Object_Ref;

  function Restore (Params   : Non_Volatile_Params_Type'Class;
                    Loading  : Loading_Operation_Type'Class;
                    Saving   : Saving_Operation_Type'Class;
                    Deletion : Deletion_Operation_Type'Class;
                    Logging  : Kind_Of_Logging;
                    Update   : Kind_Of_Update)
)
    return Memory_Object_Ref;

  Object_Exists, Object_Non_Existant, Object_Corrupt,
  Storage_Device_Error : exception;
```

Figure 11.9: Obtaining a Memory Object from the Cache Manager

11.2.3 Procedural Interface

The basic interface to transactions is the procedural interface presented in section 9.4 on page 119. Its implementation in Ada 95 is straightforward. The five procedures are grouped together in the package `Procedural_Transaction_Interface`, which is presented in figure 11.10.

Note the two overloaded operations `Begin_Transaction` and `Join_Transaction`. The versions with a string parameter allow the programmer to start respectively join a *named transaction* (see section 4.6.2 on page 59). The optional parameter `Number_Of_Participants` of the operations `Begin_Transaction` and `Begin_Or_Join_Transaction` allows the task that starts a transaction to limit the maximum number of participants.

The package also defines a set of exceptions that might be raised if the application programmer violates one of the rules of open multithreaded transactions ①. A task is, for instance, not allowed to vote on the outcome of a transaction in which it does not participate. In this case, the `No_Current_Transaction` exception is raised. Unfortunately, the Ada exception model does not provide a mechanism to attach exceptions to specific subprograms. Therefore, all possible exceptions have been declared globally inside the package, and comments have been used to state which exceptions can potentially be raised by which subprogram.

```
package Procedural_Transaction_Interface is

  procedure Begin_Transaction (Number_Of_Participants : Natural := 0);
  procedure Begin_Transaction (Name : String;
                               Number_Of_Participants : Natural := 0);

  procedure Join_Transaction (Task : Task_Id);
  procedure Join_Transaction (Name : String);
  --  Might raise No_Current_Transaction, Already_Joined_Transaction,
  --  Already_Part_Of_Other_Transaction, Transaction_Closed
  --  Transaction_Already_Committed, Transaction_Abort

  procedure Begin_Or_Join_Transaction (Name : String);
                    Number_Of_Participants : Natural := 0);
  --  Raises Already_Joined_Transaction, Transaction_Abort,
  --  Already_Part_Of_Other_Transaction, Transaction_Already_Committed

  procedure Close_Transaction;
  --  Raises No_Current_Transaction, Transaction_Abort

  procedure Abort_Transaction;
  --  Might raise No_Current_Transaction

  procedure Commit_Transaction;
  --  Might raise No_Current_Transaction, Transaction_Abort

  No_Current_Transaction, Already_Joined_Transaction,
  Already_Part_Of_Other_Transaction, Transaction_Closed,
  Transaction_Already_Committed, Transaction_Abort : exception; (1)

end Procedural_Transaction_Interface;
```

Figure 11.10: Procedural Interface for Ada 95

As mentioned in the general discussion in section 9.4.1, the procedural interface is very flexible, but it is impossible to enforce all rules of open multithreaded transactions by construct. It is, for instance, not possible to guarantee that all participants vote on the outcome of a transaction, or that all unhandled exceptions crossing the transaction boundary will be detected.

To properly use the procedural interface, the programmer must follow certain programming conventions. For each new transaction, a new Ada block must be created as shown in figure 11.11 (1). Just after the `begin` statement of the block, the programmer must make a call to `Begin_Transaction` or `Join_Transaction` (2). From then on, the task can work on behalf of the transaction. Just before leaving the block, the `Commit_Transaction` procedure must be called (3).

The Ada block must have an associated exception handling block that handles all internal exceptions (4). If recovery has been performed successfully, then `Commit_Transaction` can be called from within the exception handler (5). Otherwise, `Abort_Transaction` must be called and an external exception raised (6). A default exception handler must be added at the end of the block that intercepts any unhandled exceptions, calls `Abort_Transaction` and then propagates the exception to the outside (7).

Unfortunately this is still not sufficient to guarantee correct exception handling. Additional exceptions might occur during recovery, i.e. in the exception handling block itself. These exceptions can not be handled in the same block, and will therefore propagate to the outside, crossing the transaction boundary unnoticed. This problem can only be solved by opening new blocks inside each exception handler, with at least a default handler that aborts the transaction.

```
begin (1)
   Begin_Transaction; (2)
   -- perform work
   Commit_Transaction; (3)
exception (4)
   when ...
      -- handle internal exceptions
      Commit_Transaction; (5)
   when ...
      -- handle internal exceptions
      Abort_Transaction;
      raise External_Exception; (6)
   when others => (7)
      Abort_Transaction;
      raise;
end;
```

Figure 11.11: Programming Guidelines for the Procedural Interface

With the procedural interface it is also not possible to detect deserters, i.e. tasks that disappear due to asynchronous transfer of control. If, for instance, the task executes a transaction as part of a timed select statement (see section 10.3.6 on page 139), the `Commit_Transaction` or `Abort_Transaction` procedures will never be called if the time-out occurs inside the transaction.

11.2.4 Object-Based Interface

Most of the drawbacks of the procedural interface do not appear in the object-based interface (see section 9.4.2 on page 121). The object-based interface associates the lifetime of a transaction with the lifetime of an object, or, in case of a transaction with multiple participants, with the lifetime of a group of objects.

The object-based interface for Ada 95 is shown in figure 11.12. To start a transaction, the programmer must declare an object of type `Transaction`. The transaction begins when the object declaration is elaborated, and ends once the object goes out of scope, i.e. at the end of the block containing the object declaration. This effect is achieved by making the `Transaction` type controlled. The `Initialize` procedure starts the transaction, the `Finalize` procedure ends it. Ada requires objects to be declared inside a declarative region, which means that the programmer is forced to either associate the transaction with an already existing block, e.g. a procedure or function body, or open a new block for the transaction. This block is then at the same time transaction and exception context.

The primitive operation `Close_Transaction` allows a participant to close an active transaction. In order to vote commit, a participant must call the primitive operation `Commit_Transaction` before the end of the block. Otherwise, the task is considered a deserter, and the transaction is aborted from within the `Finalize` procedure of the `Transaction` object.

```
package Object_Based_Transaction_Interface is

   type String_Ref is access all String;
   type Task_Id_Ref is access all Task_Id;

   type Transaction (Name : String_Ref := null;
                     The_Task : Task_Id_Ref := null;
                     Max_Participants : Natural := 0) is limited private;

   procedure Close_Transaction (T : in out Transaction);
   procedure Commit_Transaction (T : in out Transaction);

   No_Current_Transaction, Already_Joined_Transaction,
   Already_Part_Of_Other_Transaction, Transaction_Closed,
   Transaction_Already_Committed, Transaction_Abort : exception;

private

   type Transaction (Name : String_Ref := null;
                     The_Task : Task_Id_Ref := null
                     Max_Participants : Natural := 0) is new
     Ada.Finalization.Limited_Controlled with record
     Committed : Boolean := False;
   end record;

   procedure Initialize (T : in out Transaction);

   procedure Finalize (T : in out Transaction);

end Object_Based_Transaction_Interface;
```

Figure 11.12: Object-Based Interface for Ada 95

There is no need for providing a procedure `Abort_Transaction`. Voting abort is simply done by raising an external exception, e.g. `Transaction_Abort`. The `Finalize` procedure of the `Transaction` object will take care of aborting the transaction as required.

The `Join_Transaction` procedure has also been eliminated, and replaced by an optional discriminant of the `Transaction` type, identifying the task to be joined. Creating or joining a *named transaction* can be achieved by supplying an optional string as a discriminant constraint. The maximum number of participants of a transaction can optionally be specified by the `Max_Participants` discriminant. Note that only discrete or access types can be used as discriminants for a type [ISO95, 3.7 (5)]. This is the reason why instead of directly passing a `Task_Id` or `String`, an *access* to `Task_Id` or an *access* to `String` must be provided by the programmer.

Figure 11.13 shows how a set of tasks can perform an open multithreaded transaction together using the object-based interface.

A special task, the `Starter_Task` ①, starts the open multithreaded transaction by declaring the transaction object `T` in the declarative region of its body ②. Any number of `Joiner_Tasks` ③ can then join the transaction by also declaring a transaction object, passing as a discriminant a pointer to the task ID identifying the starter task ④. The starter and the joiner tasks can now work together on behalf of the same transaction. Each participant must

```
task Starter_Task; (1)

task body Starter_Task is
  T : Transaction; (2)
begin
  -- perform work on behalf of
  -- the transaction
  Commit_Transaction (T); (5a)
exception
  when ...
    -- handle internal exceptions
    Commit_Transaction (T); (5b)
  when ...
    -- raise an external exception
end Starter_Task;
```

```
task type Joiner_Task; (3)

task body Joiner_Task is
  T : Transaction (null, new Task_Id
    (Starter_Task'Identity), 0); (4)
begin
  -- perform work on behalf of
  -- the transaction
  Commit_Transaction (T); (5a)
exception
  when ...
    -- handle internal exceptions
    Commit_Transaction (T); (5b)
  when ...
    -- raise an external exception
end Joiner_Task;
```

Figure 11.13: Using the Object-Based Interface

call `Commit_Transaction` once it has completed its work (5a). Of course it is also possible to commit the transaction after having handled an internal exception (5b).

Named transactions can be used as illustrated in figure 11.14. The starting of the open multithreaded transaction is different from the previous example. There is no need for a special task that explicitly starts the transaction. Any task that wants to work on behalf of the transaction can do so, provided it knows the transaction name (1). An access to this name must be passed as a discriminant constraint when declaring the transaction object `T` (2). The first task that requests to work on behalf of the transaction effectively starts the transaction, the other tasks just join it.

```
Name : String_Ref := new String' ("Job"); (1)

task type Worker_Task;

task body Worker_Task is
  T : Transaction (Name, null, 0); (2)
  begin
  -- perform work on behalf of the transaction
  Commit_Transaction (T);
exception
  when ...
    -- handle internal exception
    Commit_Transaction (T);
  when ...
    -- raise an external exception
end Worker_Task;
```

Figure 11.14: Using Named Transactions with the Object-Based Interface

The object-based interface is a lot safer than the procedural interface. The procedural interface has to rely on the programmer to follow a set of guidelines when programming transactions. When using the object-based interface, the programmer is forced to associate a block

with a transaction. As a result, it is guaranteed by construct that every participant of a transaction votes on the outcome of the transaction. If the application programmer forgets to call the commit method of the transaction object, then, enforcing a safe approach, the transaction is aborted from within the `Finalize` procedure.

Also, there is no need for having a default exception handler. Unhandled exceptions automatically cause the transaction to abort, because the `Finalize` procedure of the transaction object is invoked when the block in which the object has been declared is left. Finally, tasks executing a transaction as part of a timed select statement are also handled correctly. If a time-out occurs and the task is interrupted while working on behalf of a transaction, the `Finalize` procedure will take care of aborting the transaction as required.

Internally, the implementation of the object-based interface is based on the procedural interface. The body of the `Object_Based_Transaction_Interface` is presented in figure 11.15. The code shows, for instance, that the `Initialize` procedure of the `Transaction` type makes calls to either `Begin_Transaction`, `Join_Transaction`, or `Begin_Or_Join_Transaction`, depending on the discriminant constraints passed to the transaction object at declaration time.

11.2.5 Object-Oriented Interface

The third interface that has been developed for Ada 95 is the object-oriented interface. The package `Open_Multithreaded_Transactions`, shown in figure 11.16, declares an abstract tagged type `Transaction`. A concrete transaction must derive from this type and add code for each participant by adding primitive operations. A task that wants to work on behalf of a transaction must create a transaction object or have access to an already existing one, and then simply call the corresponding primitive operation.

A primitive operation implementing the program code for a participant must follow the programming conventions shown in figure 11.17.

The basic idea is very similar to the one used in the object-based interface. The transaction lifetime is associated with the lifetime of a controlled object `P` of type `Participant`. Every participant operation must declare such a controlled participant object in its declarative region, and must pass a reference to the transaction object as a discriminant constraint. The reference to the transaction object can be obtained by calling the function `Self`.

This reference replaces in some sort the name of the transaction. The `Initialize` procedure of the participant object calls the `Start_Or_Join_Transaction` procedure, identifying the transaction to be started or joined by means of the passed reference. Apart from this, the structure of a participant operation resembles the one described in the object-based interface. This time, the body of the primitive operation provides the transaction and exception context.

It is not possible to provide default implementations for participant operations, since their number and signatures are not known in advance. A possible solution is to provide

```
with Procedural_Transaction_Interface;
use Procedural_Transaction_Interface

package body Object_Based_Transaction_Interface is

  procedure Close_Transaction (T : in out Transaction) is
  begin
    Close_Transaction;
  end Close_Transaction;

  procedure Commit_Transaction (T : in out Transaction) is
  begin
    Commit_Transaction;
    T.Committed := True;
  end Commit_Transaction;

  procedure Initialize (T : in out Transaction) is
  begin
    if T.Name /= null then
      Begin_Or_Join_Transaction (T.Name.all, T.Max_Participants);
    elsif T.The_Task /= null then
      Join_Transaction (T.The_Task.all);
    else
      Begin_Transaction (T.Max_Participants);
    end if;
  end Initialize;

  procedure Finalize (T : in out Transaction) is
  begin
    if not T.Committed then
      Abort_Transaction;
    end if;
  end Finalize;

end Object_Based_Transaction_Interface;
```

Figure 11.15: Implementation of the Object-Based Interface

```
package Open_Multithreaded_Transactions is

  type Transaction is abstract tagged limited private;
  type Transaction_Ref is access all Transaction'Class;

  -- add code for each participant by adding primitive operations

  Transaction_Abort, Already_Part_Of_Other_Transaction,
  Transaction_Closed, Transaction_Already_Committed : exception;

private

  type Transaction is abstract tagged null record;

  function Self (T : Transaction) return Transaction_Ref;

  type Participant (Transaction : Transaction_Ref)
    is new Ada.Finalization.Limited_Controlled with null record;

  procedure Commit_Transaction (P : in out Participant);

  procedure Initialize (P : in out Participant);

  procedure Finalize (P : in out Participant);

end Open_Multithreaded_Transactions;
```

Figure 11.16: Object-Oriented Interface for Ada 95

```
procedure Participant_Code
   (T : in out Your_Derived_Type) is
   P : Participant (Self (T));
begin
   -- perform work on behalf of the transaction
   Commit_Transaction (P);
exception
   when ... =>
   -- handle internal exceptions
   when others =>
     Abort_Transaction (P);
end Participant_Code;
```

Figure 11.17: Programming Guidelines for the Object-Oriented Interface

only one primitive operation `Execute_Participant_Operation`, that takes as a parameter an access to subprogram value which will point to the actual participant code. As a result, the programmer does not have to, and therefore can not forget to declare the participant object in the actual participant operation, since this can be done once and for all in the `Execute_Participant_Operation` procedure. On the other hand, using access to subprogram types is not very elegant and complicates parameter passing.

The advantage of the object-oriented interface is that the entire open multithreaded transaction, i.e. the program code for every participant, is grouped together inside a class. This clearly improves readability, understandability and maintainability of the transaction as a whole. All external exceptions can be grouped together in the specification of the package containing the concrete transaction type. Unfortunately, Ada does not allow a programmer to associate exceptions with primitive operations. Therefore, it is not possible to state explicitly which participant may raise which external exceptions.

Code reuse is also possible, for transactions that want to perform similar work can derive from some other transaction class, and override or add new participant methods. All forms of object-oriented reuse are possible, e.g. reusing the code of a participant operation of the parent class. One could think that there might be problems due to nested participant operation invocations, since they would result in creating a hierarchy of nested transactions. Luckily, this can be avoided. Since the identity of the transaction is represented by the identity of the transaction object, we are in a situation similar to *named transactions*. In general, a call to `Begin_Transaction` from inside a transaction results in creating a nested transaction. In this case, however, the transaction support can detect that a nested participant object declaration actually refers to the *same* transaction object already in use, and therefore does not need to create a nested transaction. Of course, subsequent calls to `Commit` must then also be ignored. Only the calls associated with the outermost participant object must be taken into account.

11.2.6 Initializing and Shutting Down the Transaction Support

Regardless of what transaction interface is used, the programmer must initialize the transaction support before starting any transaction. This is done by calling a procedure named `System_Init` shown in figure 11.18.

```
procedure System_Init;

procedure System_Init (Log_Params : in Stable_Params_Type'Class;
                       Cache_Manager : in Cache_Manager_Ref;
                       Recovery_Manager : in Recovery_Manager_Ref);

procedure System_Shutdown;

Object_Exists, Object_Non_Existant, Object_Corrupt : exception;
```

Figure 11.18: Initializing and Shutting Down the Transaction Support

The procedure defines parameters that allow the application programmer to customize the transaction support according to the application requirements. In particular, a recovery manager, a cache manager, and a storage parameter identifying the storage to be used to store the log can be specified. If no parameters are supplied, the default implementation chooses a LRU cache manager, a Redo/NoUndo recovery manager, and a mirrored file on the local disk named “log” for storing the log.

If ever the system must be brought down, the procedure `System_Shutdown` should be called. It writes all dirty transactional objects to their associated storage units, and performs all outstanding log updates.

Chapter 12:
Related Work

This section gives an overview of the state of the art in transactional systems, concentrating on the ones that offer transactions at the programming language level. Clouds [DRJLAR91] and CHORUS [LJP93] for instance, two operating systems offering transactions as the prime structuring mechanism for processes, are not discussed because they are not directly concerned with providing transaction support in a programming language.

The transactional systems reviewed in this chapter comprise Argus, Camelot / Avalon, Arjuna, Venari / ML, Transactional Drago, Isis, and PJava. To illustrate transaction-oriented middleware, the CORBA Object Transaction Service and Enterprise Java Beans are also presented.

12.1 Argus

Argus is an extension of the programming language CLU that supports the construction of fault-tolerant distributed systems [Lis88]. It was developed at MIT in the 1980's. Performance measures and interesting implementation details of Argus can be found in [Lis87].

In Argus, a distributed program is composed of a group of *guardians*. A guardian controls access to one or more resources. Figure 12.1 shows how a guardian is declared in Argus [Lis85].

Each guardian resides at a single node of the distributed system, although it can change its node of residence. A guardian can create other guardians dynamically by calling creator functions ①, specifying at which node the new guardian will reside.

```
name = guardian [parameter-decls] is creator-names (1)
  handles handler-names
  {[stable] variable-decls-and-inits} (2)
  [recover body end] (3)                    -- runs after a crash
  [background body end]
  {creator-handler-and-local-routine-definitions}
end name

name = handler [parameter-decls] returns type (4)
  {signals signal-decl} (5)                  -- external exceptions
end name
```

Figure 12.1: Declaration of *Guardians* and *Handlers* in Argus

A guardian's state is split into volatile and stable objects (2) [LS83]. State stored in stable objects survives crash failures, since these objects are periodically written to stable storage. A crash destroys all volatile objects of a guardian, and also all processes that were running at the time of the crash. After the crash, the Argus system restores the guardian's code and recovers the state of the stable objects from stable storage. Then, a special recovery process is started that runs code defined by the guardian to initialize the volatile objects (3).

The state of a guardian can only be accessed by means of remotely callable procedures called *handlers* (4), which form the interface for other guardians.

12.1.1 Transaction Model

To achieve fault-tolerance, handler invocations are performed inside a transaction, which is called an *action* in Argus terms. To implement the ACID properties, Argus provides so-called *atomic objects*. Built-in types of atomic objects include atomic arrays or atomic records. Concurrency control is based on locking, providing multiple reader / single writer semantics. Recovery is based on versioning, i.e. modifications are made on a copy of the object's state. If the action commits, the copy becomes the base version, and, if the object is stable, all data is written to stable storage. If the action aborts, the copy is discarded. [WL85] shows how user-defined atomic objects can be created that increase concurrency.

All handler invocations are automatically confined inside an action, and nested handler invocations are performed as nested actions. Committing a handler invocation is considered to be the most common case, and therefore executing a `return` or `signal` statement within the body of a handler indicates commitment. To force the action to abort, the `return` or `signal` statement can be prefixed with the keyword `abort`.

Actions are also an integral part of the Argus language. Top-level actions can be created by means of the statement `enter topaction` *body* `end`. This causes the *body* to execute as a new top-level action. Likewise, it is possible to create a subaction using the statement `enter action` *body* `end`. When the body of an action completes, it must indicate whether it is committing or aborting. The former is done by executing the `leave` statement, the latter is achieved by executing `abort leave`.

12.1.2 Concurrency

Argus implements the nested transaction model (see section 3.4.4 on page 36). A guardian can service multiple handler invocations in parallel. An individual action, although, contains only one thread of control, and therefore runs sequentially.

Nevertheless, subactions can be executed concurrently using a special language construct, the `cobegin` statement, as shown in figure 12.2. The action executing the `cobegin` statement is blocked. The program code inside the `cobegin` block is called a *coarm*. Multiple instances of the coarm will be activated, and started simultaneously as concurrent subactions. Each coarm will have local instances of the variables declared in `decl-list`. Only once all subactions have completed, the parent is released. Hence, parent actions cannot execute concurrently with their children.

```
coenter [ action | topaction ]
  { [ foreach decl-list in iter-invocation ] body } end
```

Figure 12.2: Executing Nested Actions Concurrently

12.1.3 Exceptions

Argus has a very elaborate exception model. A handler can either terminate normally or by raising an external exception, i.e. returning a signal in Argus terminology. All possible external exceptions are specified in the interface of a handler as shown in figure 12.1 ⑤.

After each handler invocation, an optional exception handling block can be added as shown in figure 12.3. Exceptions can carry data, e.g. strings ① or other parameters.

```
variable := guardian_name.handler_name (parameters)
  except when signal_name (parameters): ...
      when failure (Why: string): ... ①
      when unavailable (Why: string): ...
end
```

Figure 12.3: Exception Handling in Argus

A handler is terminated exceptionally by executing the `signal` statement. This may or may not abort the corresponding action, depending on whether the statement `abort signal` or just `signal` is used.

A very advanced form of exception handling is provided for the `coenter` construct. Each coarm can have a local exception handler attached to it. By executing the `exit` or `abort exit` statement, a coarm can commit or abort and at the same time force all other coarms to abort. This is illustrated in figure 12.4.

The example executes a read operation on all copies of a replicated database. A response from any single copy will suffice. Therefore, once a read has completed successfully, the exit will commit it and abort all remaining reads. The aborts take place immedi-

ately. In particular, it is not necessary for the handler calls to complete before the subactions can be aborted.

```
coenter
  action foreach db: db_copy in all_copies (...)
  result := db.read (...)
  exit done
end except when done: ... end
```

Figure 12.4: Pre-Emption of Sibling Actions

12.2 Camelot and Avalon

Camelot [SPB88] is a facility for distributed transaction processing running on top of the MACH operating system. It was developed at Carnegie Mellon University in the late 80's as a successor of TABS [SD+85]. Camelot has been implemented as a C library, and runs on a variety of systems.

12.2.1 Transaction Model and Concurrency

Camelot is heavily inspired by Argus. The supported transaction model is exactly the same as the one found in Argus. Concurrent execution of sibling transactions is supported, but each individual transaction is sequential, and a parent transaction is suspended until all child transactions have completed their work.

Camelot also provides a node configuration application that permits to create, delete, start, shutdown, and restart *servers*, which are the equivalent of Argus guardians. Replication of servers is not directly supported.

Avalon [EMS91] is a programming language for use with Camelot that integrates transactions into C++. The new keywords that provide transaction control map nicely to the ones defined in Argus as shown in figure 12.5.

	Avalon	Argus
starting a transaction	`start transaction {}` `start toplevel {}`	`enter action ... end` `enter topaction ... end`
starting concurrent nested transactions	`costart {...}`	`coenter ... end`
committing a transaction committing with exception	`leave` Not Provided!	`return` `signal exception`
aborting a transaction aborting with exception	`undo leave` `undo (Integer) leave`	`abort return` `abort signal exception`

Figure 12.5: Mapping Avalon Keywords to Argus

aborting concurrent siblings	`leave`	`exit`
aborting with exception	`undo leave`	`abort exit`
exception handling	`except (Integer) ...`	`except when ...`

Figure 12.5: Mapping Avalon Keywords to Argus

12.2.2 Exceptions

Exceptions are handled slightly differently in Avalon than in Argus. The major difference is that in Avalon, raising an exception always results in aborting the transaction. Also, exceptions in Avalon are integers ranging from 0 to some upper limit defined by Avalon. Numbers above this limit are used for system defined error conditions.

12.2.3 Transactional Objects

In Avalon, all objects accessed by transactions must be *atomic* to ensure their serializability, transaction consistency and persistence. The Avalon language offers atomic versions of all base C++ types, e.g. `atomic_int`, `atomic_char`, `atomic_float`. All Avalon types are allocated on a special heap. Undo and persistence features are implemented by means of this heap. Committing a transaction will, for instance, force all updated data from this heap to be written to stable storage.

Users can define their own transactional types by extending one of the base Avalon classes `Recoverable`, `Atomic` or `Subatomic` presented in figure 12.6.

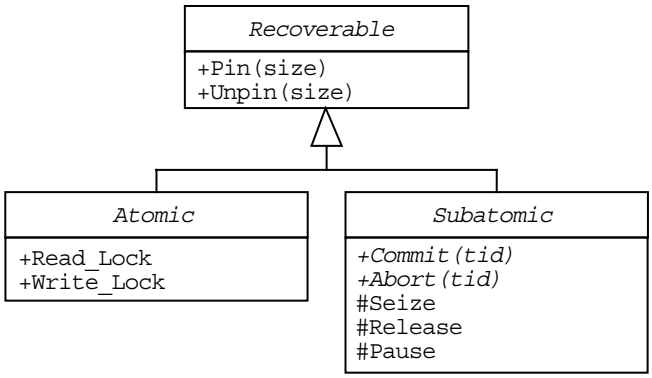

Figure 12.6: The Avalon Base-Classes

The class `Recoverable` implements persistence. It provides two public methods `Pin` and `Unpin`. Calling `Pin` ensures that subsequent changes to the object's state will not be recorded to stable storage until a later matching call to `Unpin`. The integer argument to `Pin` and `Unpin` should be the size of the object that is modified. After a crash, a recoverable object will be restored to a previous state in which it was not pinned. In order to guarantee consistency of the application, all modifications to recoverable objects must be made between calls to `Pin` and `Unpin`, or in a special pinning block `pinning () {...}`.

The class `Atomic` provides strict concurrency control, i.e. multiple readers / single writer. A programmer can derive from this class to implement application-specific atomic types. All observer methods of the new class must call the operation `Read_Lock` prior to accessing any attributes of the class. Likewise, all modifications must be preceded by a call to `Write_Lock`. Persistence for the class `Atomic` is inherited from the class `Recoverable`.

The class `Subatomic` can be used if the programmer wants to exploit the object's semantics to permit higher levels of concurrency and more efficient recovery. Since the state of a subatomic object is potentially accessed concurrently, an additional short-term mutual exclusion lock is provided for each object. This lock must be seized, released, or temporarily released by means of the protected methods `Seize`, `Release` and `Pause`. This can also be achieved by using the construct `when (`*`expression`*`) statement` or `whenswitch (`*`expression`*`) statement`.

Classes derived from the `Subatomic` class are expected to reimplement the `Commit` and `Abort` methods. Both methods carry the transaction id as a parameter. As a result, it is possible to implement concurrency control based on the so-called hybrid atomicity model [FLW92].

Even though Avalon is an extension of the C++ programming language specially designed for Camelot, the application programmer must still adhere to some programming guidelines [EMS91, chapter 22]. Care must be taken, for instance, not to allocate any atomic object on the stack by just declaring a variable of an atomic type. Instead, references to atomic objects must be used.

12.3 Arjuna

Arjuna [Shr95] is an object-oriented programming system implemented in C++ that provides a set of tools for the construction of fault-tolerant distributed applications. It has been developed at the University of Newcastle upon Tyne, UK.

The computational model of Arjuna for constructing robust distributed applications is based upon the concept of using nested transactions for controlling operations on local or remote objects. Operations on remote objects are invoked using remote procedure calls, which are implemented using a preprocessor called *Rajdoot* [SDP91]. Apart from this, Arjuna is based on standard C++ only. All functionality is implemented inside C++ classes, some of which can be customized by the application programmer using inheritance.

A Java version of Arjuna, *JTSArjuna*, that basically is an implementation of the CORBA Object Transaction Service (see section 12.8 on page 181) for Java, has been released recently [Arj00].

12.3.1 Transaction Model

Arjuna supports nested transactions with multithreading (see section 3.6.1 on page 43). Multiple threads can work on behalf of the same transaction, but threads are not coordinated on transaction commit.

The interface to transactions is based on the `AtomicAction` class [PSWL95], which provides among others the methods `Begin`, `Abort` and `End` that control the life cycle of the transaction. Figure 12.7 shows how nested transactions can be programmed in Arjuna. To ensure that nesting is correctly managed, `AtomicAction` maintains a class variable called `Current` which points to the current active transaction. To support concurrency inside a transaction, Arjuna provides the class `ConcurrentAtomicAction`, which is derived from `AtomicAction`. Unfortunately, even in [SPW+94] no example was shown that uses this feature.

```
{
  AtomicAction A, B;

  A.Begin(); // perform work on A
    B.Begin(); // work on B
    B.Abort();
  A.End(); // commit A
}
```

Figure 12.7: Nested Transactions in Arjuna

Interestingly, the recent Java implementation of Arjuna provides support for thread control inside a transaction. The Programmer's Guide [Arj00] states:

"It is possible that if one thread terminates a transaction, other threads may still be active within it. By default, JTSArjuna will issue a warning in that case. However, it will allow the transaction termination to continue. Other solutions to this problem are possible, e.g. blocking the thread which is terminating the transaction until all other threads have disassociated themselves from the transaction context."

Therefore, JTSArjuna provides the class `CheckedAction`, which allows the thread / transaction termination policy to be overridden. Each transaction has an instance of this class associated with it, and application programmers can provide their own implementations on a per transaction basis.

12.3.2 Exceptions

Exceptions are not integrated with transactions in Arjuna. Because a transaction is not an exception context, unhandled exceptions crossing the transaction boundary are not detected, and hence do not affect the status of a transaction.

Exceptions are also not used to notify the application programmer of the transaction outcome. Instead, the `End` method of the `AtomicAction` class returns an error code that must be checked by the application programmer.

12.3.3 Transactional Objects

Concurrency control in Arjuna is based on locking. The `Lock` class provides a default implementation of the multiple readers / single writer paradigm. It can be extended to provide user-defined locking. The abstract method `!=` is then used to determine if two locks conflict or not [PS88].

Transactional objects, here called persistent objects, must be derived from the class `LockManager`, and must allocate appropriate locks in their access procedures, e.g. an observer method must allocate a read lock, and pass it to the protected method `Setlock` of the `LockManager` class. For persistence, `SaveState` and `RestoreState` methods must also be implemented by each user-defined class. Details on the implementation of persistence for Arjuna can be found in [DPSW89].

12.4 Venari / ML

Venari / ML [HKM+94] is a transactional extension of Standard ML. It allows the programmer to create transactional versions of higher-order functions. It has been developed at Carnegie Mellon University, Pittsburgh, USA.

12.4.1 Transaction Model and Concurrency

In Venari / ML transactions are factored into four separable features: persistence, undoability, locking and threads. Relying on function composition, these features can be composed to build the traditional transaction models or other models with weaker semantics. Some existing transaction models are analyzed, e.g. flat transactions, nested transactions, concurrent transactions, and multithreaded transactions. Particular emphasis is placed on the nested multithreaded transaction model (see section 3.6.2 on page 45).

Given a function f, a transactional version of this function can be created by applying the `transact` function to it as shown in figure 12.8.

```
((transact f) a)
  handle Abort => [some work] (1)
```

Figure 12.8: Creating a Transaction in Venari / ML

Since `(transact f)` is simply a function, and functions are first-class “citizens”, transactions are really an integral part of the language.

12.4.2 Exceptions

Exceptions are used to inform the initiator of a transaction in case the transaction aborts. No distinctions are made between internal or external exceptions. The `Abort` exception

handler ① allows some special action to be taken if the transaction aborts. Unhandled exceptions crossing the transaction boundary cause the transaction to be aborted.

12.5 Transactional Drago

Drago [MAAG96] is a language developed in Spain at the Technical University of Madrid and at the University of Las Palmas de Gran Canaria. It has been designed and implemented as a fault–tolerant, distributed extension of Ada 83. Fault tolerance is achieved by active replication of virtual nodes, which are called *agents* in Drago. Agents are very similar to Ada tasks (see section 10.3.1 on page 134). They provide entries in their interfaces. The system ensures that all replicated agents accept entries in the same sequence.

Replicas in Drago form groups. All replicated agents must behave in a deterministic manner [GMAA97]. Drago also exploits groups as a naming abstraction by offering so–called cooperative groups whose member agents may be different and may behave non–deterministically.

12.5.1 Transaction Model

Drago has been recently extended to include transactions [PMJPA98]. *Transactional Drago* supports multithreaded transactions (see section 3.6.2 on page 45), i.e. tasks are forked and joined at the transaction boundaries.

Transactions are provided to the programmer by means of the transactional block statement `begin transaction` / `end transaction`. It can be used wherever a programmer is allowed to use the normal Ada block statement. A transactional block has the same structure than a normal block, e.g. it has a declaration part. Concurrency inside a transaction is achieved by declaring tasks in the declaration part of a transactional block. If tasks want to synchronize, they must do so explicitly using the Ada rendezvous concept. The transaction completes once all internal tasks have terminated.

Data declared inside a transaction are subject to automatic atomicity and concurrency control. Transactional concurrency control is based on read / write locking.

Transactional Drago extends the group concept of Drago to cover transactional groups. Transactional groups can be either cooperative or replicated. Transactional groups must be called from within a transaction and may themselves only call other transactional groups. Transactional agents handle requests from the same client serially, but requests from different clients are handled concurrently.

Transactional Drago code is translated into Ada via a preprocessor. The generated code makes calls to the TransLib framework [JPPMA00].

12.5.2 Exceptions

Transactional Drago integrates exceptions and transactions, i.e. transactions are exception contexts. Implicit transaction aborts due to crashes or deadlocks are signalled using the predefined exception `Transaction_Abort_Error`. Any unhandled exception that crosses a transaction boundary results in aborting the transaction. Therefore, explicit aborts can be achieved by raising an exception.

Since multiple tasks are allowed to work concurrently inside a transaction, multiple exceptions can occur concurrently. The system resolves exceptions raised concurrently by several participating tasks before signalling a resulting exception to the outside of the transaction. In a distributed setting, exception resolution is split into a local resolution phase and a distributed resolution phase.

Transactional Drago does not allow external exceptions to be declared in the transaction interface, therefore no special distinction is made between internal and external exceptions.

12.6 PJama

The PJama project, a collaboration between the University of Glasgow and Sun Microsystems Laboratories formerly known as the PJava project [ADJ+96, AJDS96], aims at providing *orthogonal persistence* (see section 2.5 on page 25) for the Java language without modifying the language.

The system allows a programmer to define *roots of persistence*, where individual objects can be registered during run-time. All objects reachable from a persistent root are made persistent (*persistence by reachability*). This is achieved by modifying the Java Virtual Machine.

12.6.1 Transaction Model

The design document of PJama [ADS96] also mentions support for transactions. [Day96] presents a detailed design for extensible transaction management in PJama. Just as persistence, transactions in PJama are transparent to the application, i.e. any normal Java class can be used in a transaction. Again, this can only be achieved by modifying the Java Virtual Machine. Classes that want to start or end a transaction are of course transaction aware, and can therefore not be used with a non-modified Java Virtual Machine.

The concurrency model supported by PJama transactions seems to be close to the multithreaded transaction model (see section 3.6.2 on page 45). The transaction model itself is not imposed, though. The design leaves the transaction model open, by defining a class `TransactionShell` that can be extended in order to provide support for different transaction

models, e.g. flat transactions, nested transactions, split transactions (see section 3.4.5 on page 38) or SAGAS (see section 3.4.8 on page 40).

Transactions are defined by creating an instance of a subclass of the class `TransactionShell`. The programmer chooses the subclass depending on the transaction model that suits the application requirements. The constructor of the subclass associates a `Runnable` object with the transaction. The real transaction is actually created when the `start` method is invoked on the `TransactionShell` object, which results in executing the `run` method of the `Runnable` object in a transactional manner.

12.6.2 Exceptions

[Day96] mentions that the transaction framework confines exceptions that are uncaught by a transaction body to the limit of the transaction. Unfortunately, no further details on exception handling are given.

12.7 Isis

Isis [BvR94] was the first group communication system based on the notion of view synchrony. It offers the abstraction of a group together with operations for determining group membership and a set of multicast communication primitives with various semantics.

Isis is known in the first place for this model of group communication, but the Isis toolkit also includes a transaction manager built on top of the group abstraction [GBCvR93]. Birman describes in [Bir85] the use of replication and transactions for building reliable distributed applications. Although Birman never clearly describes his assumptions about the computational model, it can be deduced that replicas may be multithreaded, but that each transaction itself must behave deterministically, otherwise, the recovery protocol described in [Bir85] would not work correctly. Isis' transaction toolkit supports nested transactions; remote calls are viewed as nested transactions. The application is offered constructs to declare new top–level transactions and to abort voluntarily.

12.8 CORBA Object Transaction Service

The *Common Object Request Broker Architecture* [Obj95], *CORBA* for short, is a middleware that specifies the way in which distributed objects communicate on a network, and how clients are to invoke methods on these objects, without knowing the locations and particular execution environments of the objects. Just as UML, CORBA is a standard approved by the Object Management Group.

CORBA is programming language independent, for it uses an interface definition language called IDL to specify the interfaces among objects. Mappings exist from IDL to all

commonly used programming languages. The communication between objects is provided by the Object Request Broker, the ORB. ORB implementations for different languages and platforms can work together using the Internet Inter-ORB Protocol, IIOP for short.

The CORBA standard further defines an elaborate set of standard object services, such as the Naming Service, the Persistence Service, the Concurrency Service, the Event Service, the Trading Service, and also the *Object Transaction Service* [Obj00], OTS for short. A new revision of the OTS has been released in May 2000.

The Object Transaction Service provides an object-oriented framework for distributed transaction processing. It defines CORBA IDL interfaces that allow multiple distributed objects to participate in a transaction. OTS conforms to the X/Open DTP Reference Model and defines the integration of transactional subsystems using X/Open APIs.

12.8.1 Transactional Objects

Any CORBA object whose behavior is affected by being invoked within the scope of a transaction must support the *Transactional Object* interface. A CORBA object whose state is affected by committing or rolling back a transaction is called a *Recoverable Object*. A recoverable object must participate in the Object Transaction Service protocols, in particular in the two-phase commit protocol by registering a *Resource* object with the transaction. The resource interface comprises the `prepare`, `rollback`, `commit` and `commit_one_phase` methods.

12.8.2 Transaction Model

Any OTS implementation must provide support for flat transactions (see section 3.4.1 on page 33). Support for nested transactions is optional. OTS allows multiple threads to be associated with the same transaction context [Obj00, 1.2.5.2], which means that concurrency inside a transaction is allowed (see section 3.6.1 on page 43). However, just as there is no requirement for an ORB to service multiple requests in parallel, there is also no requirement for CORBA Recoverable Objects to be thread safe. It is therefore the responsibility of the application programmer to implement the resources according to the way the application makes use of them, e.g. to protect them from simultaneous accesses.

Transaction control in CORBA OTS is provided by means of the `Current` interface. It defines the basic operations `begin`, `commit`, `rollback`, `rollback_only`. More elaborate operations allow the programmer to query the status of a transaction (`get_status`), obtain a string describing the transaction (`get_transaction_name`), or even set time-outs for the transaction (`set_timeout`). A thread can temporarily suspend its work on behalf of a transaction by calling `suspend`. As a result, a `Control` object is handed back to the thread, which can later on be used to reestablish the transaction context by calling `resume`.

If such a `Control` object, which can also be obtained by calling `get_control`, is handed over to other threads, they can also be associated with the transaction by calling

`resume`. No check is made to verify that the thread is not already working on behalf of some other transaction. The previous transaction context of the thread is discarded.

The revision of the CORBA OTS in May 2000 also adds the possibility to register a synchronization object with a transaction. The synchronization object will be notified before the transaction commits, and after completion of the commitment.

12.8.3 Exceptions

The CORBA OTS does not integrate exceptions and transactions; transactions are not exception contexts. However, exceptions are used to notify clients of the outcome of a transaction, e.g. `Transaction_Required`, `Transaction_Rolledback`, `Invalid_Transaction` or `Subtransactions_Unavaiable`.

12.9 Enterprise Java Beans

Enterprise Java Beans is a higher-level component-based architecture for distributed business applications [VR99]. The most recent version of the EJB specification released by Sun Microsystems is version 1.1 [SHM+00]. Version 2.0 is close to completion.

EJB aims at simplifying the development of complex systems in Java by dividing the overall development process into six different architecture roles that can be performed by different parties.

One of the architecture roles is the *Enterprise Bean Provider*. Typically performed by an application-domain expert, e.g. from the financial industry, the enterprise bean provider builds a component, called an *enterprise bean*, that implements the business methods without being concerned about the distribution, transaction, security, and other non-business-specific aspects of the application.

The *EJB Container Provider* on the other hand is an expert in distributed systems, transactions and security. The container provider must deliver tools for the deployment of enterprise beans, and a run-time system that provides the deployed beans with transaction and security management, distribution, management of resources, and other services.

The other architecture roles are the *EJB Server Provider*, the *System Administrator*, the *Deployer*, and finally the *Application Assembler*[1].

12.9.1 Session Beans and Entity Beans

The EJB architecture defines two types of enterprise beans, *session beans* and *entity beans*.

The *session bean* can be transaction-aware, may update shared data in an underlying database, and is relatively short-lived. It can, for instance, be removed by a client, or be destroyed due to a time-out. Session beans also disappear if the EJB container crashes. In

1. Version 2.0 of the EJB specification also defines the *Persistence Manager Provider* role.

general, session beans contain state. However, the EJB specification also defines a *stateless session bean*.

Entity beans provide an object view of data in a database and offer shared access to this data by multiple users. They are long-lived, i.e. their lifetime is not linked to the lifetime of the container, but usually to the lifetime of the underlying database. Therefore, the entity bean itself, the primary key with which the state of the entity bean can be retrieved from the database, and its remote references survive a crash of the EJB container.

Structural and assembly information necessary for embedding a bean in an application is contained in the *Deployment Descriptor* of a bean. Besides the name, class and interfaces of the bean, it also contains additional information, e.g. if bean-managed or container-managed persistence is used for the entity bean.

12.9.2 Transaction Model

The Enterprise Java Beans architecture supports flat transactions only (see section 3.4.1 on page 33). This decision was mainly motivated by the fact that in general only flat transactions are supported by existing transaction processing and database management systems. If vendors provide support for nested transactions in the future, Enterprise Java Beans may be enhanced to take advantage of nested transactions.

The methods of an enterprise bean do not handle transactions directly. Instead, the Deployment Descriptor determines the transactional behavior of a bean or a method. Possible transaction policies are presented in figure 12.9.

Policy	Meaning
`TX_NOT_SUPPORTED`	The method can not be called from inside a transaction.
`TX_SUPPORTED`	The method can be called from inside a transaction.
`TX_MANDATORY`	The method must be called from inside a transaction. If this is not the case, an exception is thrown to the caller.
`TX_REQUIRED`	The method requires to be executed from inside a transaction. If this is not the case, a new transaction is created.
`TX_REQUIRED_NEW`	The container creates a new transaction before executing the method.
`TX_BEAN_MANAGED`	The session beans are allowed to manage transactions explicitly by calling `javax.transaction.CurrentTransaction`.

Figure 12.9: Enterprise Java Beans Transaction Policies

Multiple threads are allowed to work on behalf of the same transaction, but some important restrictions, presented in the next section, apply.

12.9.3 Concurrency Control

Inter-Transaction Concurrency

When writing the entity bean methods, the bean provider does not have to worry about concurrent accesses by multiple transactions. The bean provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently by multiple transactions.

The EJB specification mentions two different implementation strategies. The container can activate multiple instances of a bean, one for each transaction, and let the underlying database handle proper serialization. Depending on what kind of lock the `ejbLoad` method acquires, this may unnecessarily block read-only transactions, or lead to deadlocks. The other solution is to activate only a single instance of the entity bean, and serialize the accesses by multiple transactions to this instance, which also restricts concurrency among transactions dramatically.

Intra-Transaction Concurrency

The EJB transaction model allows multiple threads to be associated with the same transaction context. However, concurrent calls in the same transaction context to the same entity object are illegal. Only *loopback calls*, i.e. a call that spreads from an object A to an object B and then back to A, are allowed. In such a case, the thread executing the first operation on A is suspended. A second thread is needed to execute the second operation on A. Thus, loopback calls require a certain form of concurrency.

A bean provider can specify if an entity bean is *reentrant* or *non-reentrant*. If an instance of a non-reentrant entity bean executes a client request in a given transaction context, and another request with the same transaction context arrives for the same entity object, the container will reject the second request by throwing the `RemoteException` exception. In this case, loopback calls are also prohibited. Reentrant beans on the other hand allow loopback calls, but since the container can not distinguish a loopback call from a concurrent call, the client programmer must be careful to avoid code that could lead to a concurrent call in the same transaction context.

Transaction “diamond” scenarios, e.g. a program A that calls program B and C from within the same transaction, where B and C both access the same entity bean D, are also problematic. [SHM+00, EJB.11.7] states that a container must provide support for local diamonds, a situation that occurs if A, B, C, and D are deployed in the same EJB container. Distributed diamond support is not required. However, an implementation supporting dis-

tributed diamonds must provide a consistent view of the state of an entity bean within a transaction.

12.9.4 Exceptions

The EJB specification defines precise rules for exception handling [SHM+00, EJB.12]. Two classes of exceptions are defined: *application exceptions* and *system exceptions*.

Application exceptions can be defined by the bean provider in the throws clauses of the methods of a bean. Application exceptions are intended to be handled by the client, and therefore should be used for reporting business logic exceptions. Application exceptions do not automatically result in aborting the transaction, and therefore the bean provider must ensure that the instance of the bean is in a consistent state before raising the exception, or explicitly mark the transaction context for rollback by calling `EJBContext.setRollbackOnly()`.

System exceptions represent situations that prevent a method from successfully completing, e.g. failure to obtain a database connection, Java Virtual Machine errors, or an unexpected `RuntimeException`. If a bean encounters such a situation and it does not know how to recover from it, the method should throw a suitable system exception, which must be a `RuntimeException`, or a subclass of `javax.ejb.EJBException`. The EJB container catches all system exceptions, logs them, aborts the current transaction, and throws `java.rmi.RemoteException` to the calling client. The container also ensures that no other method will be invoked on the bean instance that threw the system exception.

Part IV

Case Study

Chapter 13:
Online Auction System

This chapter presents a case study intended to show how open multithreaded transactions can be used for designing and structuring complex, distributed, and fault-tolerant systems. The case study is about an auction system, a typical distributed application as found in the growing field of e-commerce.

13.1 Requirements

The informal description of the auction system presented in this section is inspired by the auction service example presented in [Vac00], which in turn is based on auction systems found on various internet sites, e.g. `www.ebay.com`, `www.ubid.com` or `www.ibazar.com`.

13.1.1 General Requirements

The auction system runs on a set of host computers connected via a network. Clients will access the auction system from one of these computers.

The system allows the clients to buy and sell items by means of auctions. Different types of auctions are supported, namely *English auctions, Dutch auctions, 1st Price auctions, 2nd Price auctions*, etc.

The *English auction* is the most well-known form of auction. The item for sale is put up for auction starting at a relatively low minimum price. Bidders are then allowed to place their bids until the auction closes. Sometimes, the duration of the auction is fixed in advance, e.g. 30 days, or, alternatively, a time-out value can be associated with the auction.

Each time a new bid is registered, the time-out is reset. The auction closes once the time-out expires.

In a *Dutch auction*, the starting price is set to a high price. Then, following a predefined interval, e.g. once per day, this price is lowered by a certain amount. The first bidder wins the auction.

During a *1st Price auction*, all bidders place one secret bid. When the auction closes after a specified amount of time, the bidder that made the highest bid wins the auction. The *2nd Price auction* is based on the same principle. However, the winner, i.e. the bidder that placed the highest bid, must pay only the amount of the next best bid.

13.1.2 Registration

Any client interested in using the auction system services must first register with the system by filling out a *registration form* on which he or she must provide his or her real name, postal address and email address, and a desired username and password.

Moreover, all registered users must deposit a certain amount of money or some other security with the auction system at registration time. The money is transferred to a bank account under control of the auction system. When bidding for goods, the sum of the bids placed by a client may never exceed the money available on his or her account.

Once the registration process is completed, the client becomes a *member* of the auction system.

13.1.3 Login

A member of the auction service that wants to make use of the system must first login to the system using his or her username and password. Once logged, the member may choose from one of the following possibilities:

- Start a new auction,
- Browse the current auctions,
- Participate in an ongoing auction,
- Consult the history of other members, or
- Deposit or withdraw money from his or her account.

13.1.4 Starting an Auction

A member wanting to start a new auction must fill out an *item form* describing the item to be put up for auction. Required information includes a title, a detailed description of the item the member wants to sell, and an opening bid. In addition, the type of auction to be used must be specified.

Once the item form has been submitted successfully, the system starts the auction and inserts it into the list of current auctions.

13.1.5 Browsing the List of Current Auctions

Any member logged into the auction system is allowed to browse the list of current auctions. The information available in the current auction list is the title of the auction, the auction type, the description of the item for sale and the expiration date of the auction.

13.1.6 Participating and Bidding in an Auction

While browsing the list of current auctions, a member can decide to participate in one or several of them. To bid on an item, a participant simply has to enter the amount of the bid. A valid bid must fulfill the following requirements:

- The amount left on the bank account of the member that wants to place a bid is at least as high as the sum of all his or her pending bids plus the new bid. This requirement ensures that a member is always in the position to pay for all items he or she placed bids on.
- The member placing the bid is not the member having started the auction. This rule prohibits a seller to bid in his or her own auction.
- The auction has not expired.
- In *English auctions*, the new bid must be higher than the current bid. If nobody has placed a bid yet, then the bid must be at least as high as the opening bid.
- In *Dutch auctions*, the new bid is usually equal to the current bid. In general, bids that are higher than the current bid are also accepted.
- In *1st Price auctions* and *2nd Price auctions*, the new bid must be at least as high as the opening bid.

If any of the previously stated requirements is not met, the auction system rejects the bid.

13.1.7 Closing an Auction

The time of closure of an auction depends on the type of auction (see section 13.1.1 on page 189). If an auction closes, and no participant has placed a valid bid, then the auction was unsuccessful. In that case, the auction system does not charge any money for the provided services.

If the auction closes and at least one valid bid has been made, then the auction ends successfully. In that case, the participant having placed the highest bid wins the auction. The money is withdrawn from the account of the winning participant and deposited on the account of the seller, minus two percent, which is deposited on the account of the auction system for the provided services.

13.1.8 Member History

The auction system keeps track of all auctions started or won by a member. Any member can consult the history of other members.

13.1.9 Delivery of the Goods

The auction site `Ibazar`, for instance, trusts its members to effectively send the goods that have been sold in an auction to the winning member. In these systems, the winning member can, once he or she has received the item, vote on the quality of the delivery. This vote will be registered in the history of the seller.

Other systems provide a special escrow service that blocks the money of the winning bidder until the seller sends the goods. Only when the goods have been received and the bidder is satisfied, the money gets transferred to the seller account.

13.1.10 Fault-Tolerance Requirements

The auction system must be able to tolerate failures. Crashes of any of the host computers must not corrupt the state of the auction system, e.g. money transfer from one account to the other should not be executed partially. Temporary unavailability is acceptable.

13.2 Application Design

The auction system is an example of a dynamic system with cooperative and competitive concurrency. The concurrency originates from the multiple connected members, who each may participate in or initiate multiple auctions simultaneously. Inside an auction, the members cooperate by bidding for the item on sale. On the outside, the auctions compete for external resources, such as the user accounts. The system must be dynamic, since a member must be able to join an ongoing auction at any time.

To deal with this complexity, the design of the auction system is based on open multi-threaded transactions.

13.2.1 Transactional Objects in the Auction System

Any data used from within a transaction and also any data that should survive crashes must be encapsulated inside a transactional object.

The transactional objects needed for implementing the auction system are fairly easy to identify. Every concrete transactional object must implement the operations `Create`, `Restore` and `Delete`. The relationships among the classes are presented in figure 13.1 (for brevity, `T_` stands for `Transactional_`).

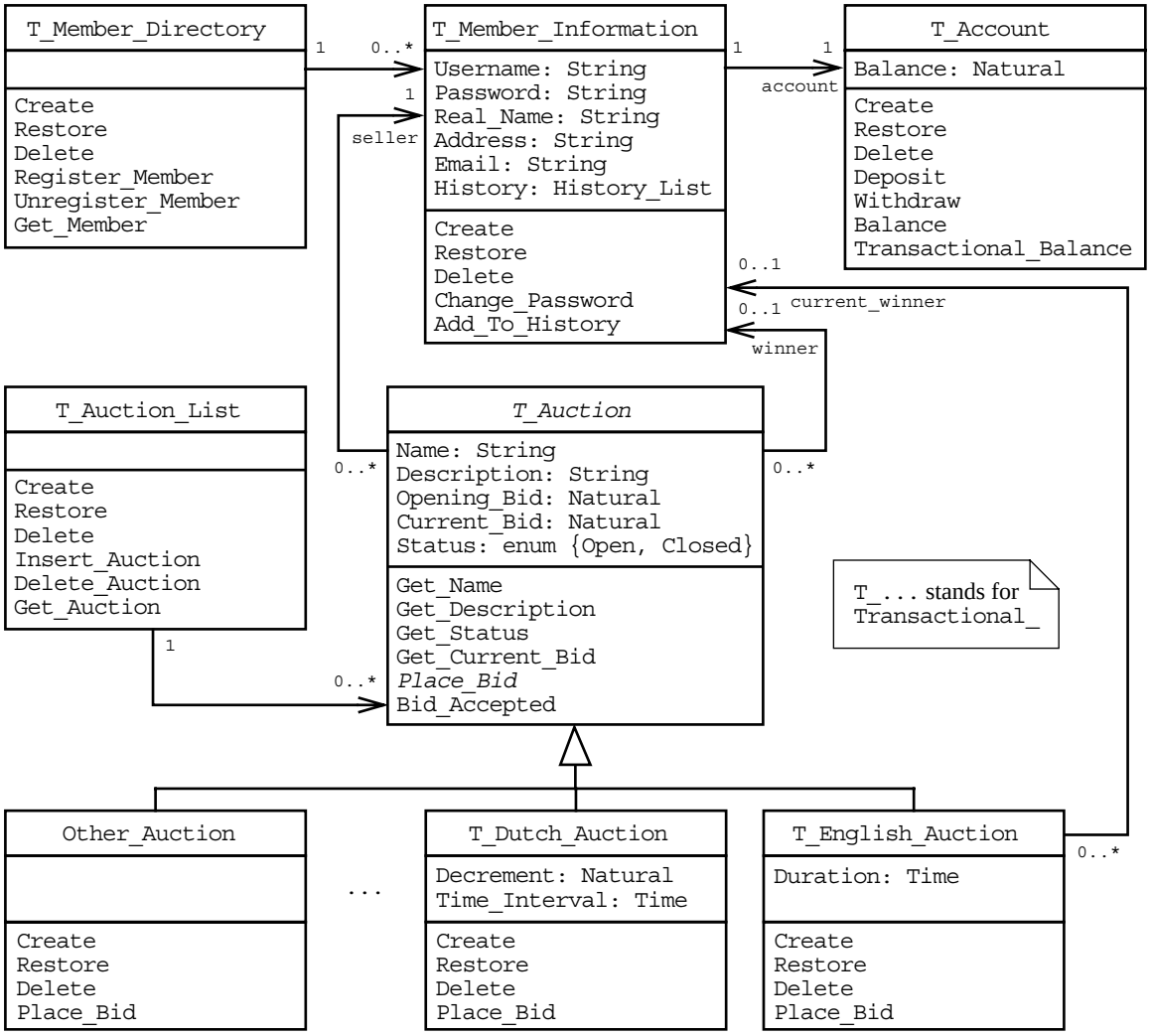

Figure 13.1: Transactional Objects found in the Auction System

Registration and history information of members are stored in the `Transactional_Member_Information` class. In addition to providing a constructor method `Create`, the class also offers a method that allows a member to change password, and a method `Add_To_History` that allows one to append a successful auction to the history of a user.

The `Transactional_Member_Directory` class defines the set of all registered members. It provides methods to register and unregister members, and to retrieve the `Transactional_Member_Information` object of a member.

The abstract `Auction` class represents auctions. It defines methods to query information about the auction, and a method that can be called by a bidder to know if a previously placed bid has been accepted. The `Auction` class has several concrete subclasses, e.g. `English_Auction` and `Dutch_Auction`, that define the different auction types. Every subclass must implement the required operations `Create`, `Restore` and `Delete`, plus the `Place_Bid` operation.

The Auction_List class contains the list of all current auctions. It provides the usual operations available on lists, i.e. Insert_Auction, Remove_Auction and Get_Auction.

Finally, every member has a bank account, represented by the Account class.

13.2.1.1 The Account Class

The description of the Transactional_Account class, which encapsulates a member's bank account, is given in more detail, because section 13.3.1 on page 198 shows how the class can be realized in Ada using the Ada implementation of the OPTIMA framework.

The attributes and methods of the non-transactional Account class are presented in figure 13.2. The Deposit, Withdraw and Balance operations have the usual semantics. A Withdraw is only possible if there is enough money on the account, since members are not allowed to overdraw their account. If this is not the case, the exception Not_Enough_Funds is raised.

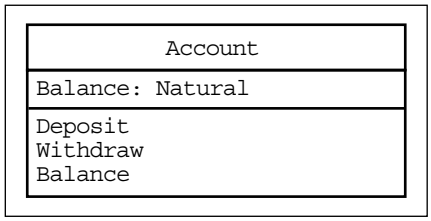

Figure 13.2: The Account Class

13.2.1.2 The Transactional_Account Class

The Transactional_Account class (see figure 13.1) provides additional Create, Restore and Delete operations, plus a Transactional_Balance operation. This operation is similar to the Balance operation, but it allows one to query the current balance of the account even if other active transactions have modified the balance. Such an operation is necessary, for it allows the owner of the account to query how much money is available for new bids in case his or her other bids placed in other auctions are accepted.

Strict concurrency control designates Withdraw and Deposit as *modifiers*, Balance and Transactional_Balance as *observers*. Unfortunately, for accounts, strict concurrency control unnecessarily restricts concurrency. Analyzing the semantics of the operations of the Transactional_Account class reveals that some of them commute (see section 7.4.2.1 on page 85).

The compatibility table for the operations of the Transactional_Account class is given in figure 13.3. It is based on backward commutativity.

	Deposit(y)	Withdraw(y)	Balance	Trans_Bal
Deposit(x)	yes	yes	no	yes
Withdraw(x)	no	yes	no	yes
Balance	no	no	yes	yes
Trans_Bal	yes	yes	yes	yes

Figure 13.3: Compatibility Table for the Transactional_Account Class

Note that the table is not symmetric. A `Deposit` operation commutes with a `Withdraw` operation, but `Withdraw` does not commute with `Deposit`. This is due to the fact that the `Withdraw` operation can not be completed successfully if there is not enough money on the account. An uncommitted `Deposit` operation could give the illusion that a withdraw is possible, but if the deposit is rolled back later on, the withdraw would not be valid anymore.

The `Transactional_Account` is an example of a *self-checking transactional object* (see section 4.6.4.2 on page 60). If a withdraw can not be completed, the exception `Not_Enough_Funds` is raised. The participant of the open multithreaded transaction that invoked the `Withdraw` operation is forced to address this abnormal situation by providing a local exception handler. Otherwise, the exception crosses the transaction boundary and the transaction is aborted.

13.2.2 Open Multithreaded Transactions in the Auction System

Open multithreaded transactions are used throughout the design of the auction system according to the following rules:

- Any operation that might potentially interfere with other operations executed concurrently must be encapsulated inside a transaction.
- Any set of operations that should never be executed partially must be encapsulated inside a transaction. This includes also creation of several transactional objects that logically belong together.
- Any set of operations that might have to be undone must be encapsulated inside a transaction or subtransaction.

Threads that want to cooperate by accessing the same transactional objects must be participants of the same open multithreaded transaction.

The following two sections present the design of two open multithreaded transactions found in the auction system. The *Registration* transaction uses a single-threaded, non-nested transaction, whereas the *English auction* transaction is based on a multithreaded, nested transaction.

13.2.2.1 Registration Transaction

A client wanting to become a member of the auction system must first register with the system (see section 13.1.4 on page 190) by filling out the *registration form*. As a consequence, a new `Account` with the initial deposit is created for the member. Then, a new `Member_Information` object is created, initialized with all relevant data from the registration form and a reference to the new account, and finally inserted into the `Member_Directory`. These three operations, namely creating the `Account` and `Member_Information` objects and updating the `Member_Directory`, must be performed atomically, since a partial execution,

e.g. creating the `Member_Information` object without registering it in the `Member_Directory`, would lead to permanent storage leaks.

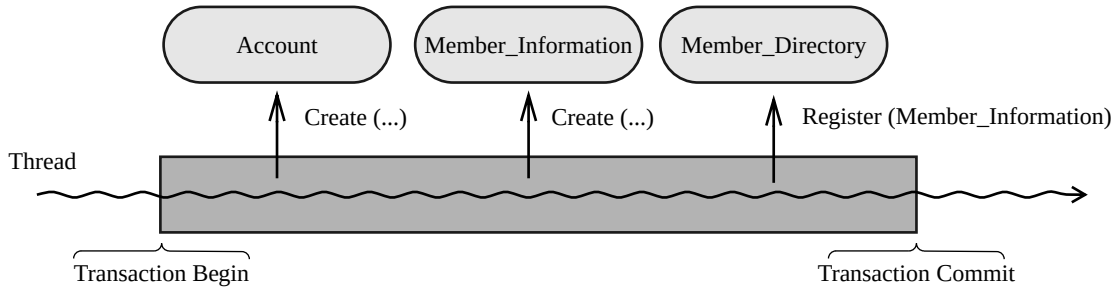

Figure 13.4: The *Registration* Transaction

In order to prevent this from happening, the two `Create` operations and the `Register` operation are executed in a transaction as shown in figure 13.4. In this case, the structure of the open multithreaded transaction is identical to the one of a flat transaction, namely single-threaded, without subtransactions.

Section 13.3.3.1 on page 205 shows how the registration transaction has been implemented using the object-based interface for Ada 95.

13.2.2.2 English Auction

Maybe the most important requirement for auctions is that they must be fault-tolerant. All-or-nothing semantics must be strictly adhered to. Either there is a winner, and the money has been transferred from the account of the winning bidder to the seller account and the commission has been deposited on the auction system account, or the auction was unsuccessful, in which case the balances of the involved accounts remain untouched.

Auctions are complicated interactions among multiple participants. They incorporate cooperative and competitive concurrency. The participants of an auction cooperate by placing bids on the same item. Members are allowed to participate in several auctions at the same time. Concurrently executing auctions compete for the money on the member accounts.

The number of participants of an auction is not fixed in advance. Therefore, auctions must also be dynamic: new participants must be able to join the auction at any time.

All these requirements can be met if an individual auction is encapsulated inside an open multithreaded transaction. A graphical illustration of an *English auction* is shown in figure 13.5.

Every member creates a new thread that represents the member inside the auction transaction. As a result, members can participate in multiple auctions at the same time.

In figure 13.5, member 1 starts a new auction, creating a new seller thread. Once the item form has been completed, a new open multithreaded transaction, here named `T1`, is started. Then, the seller creates a new auction object by invoking the `Create` function, and inserts it into the list of current auctions.

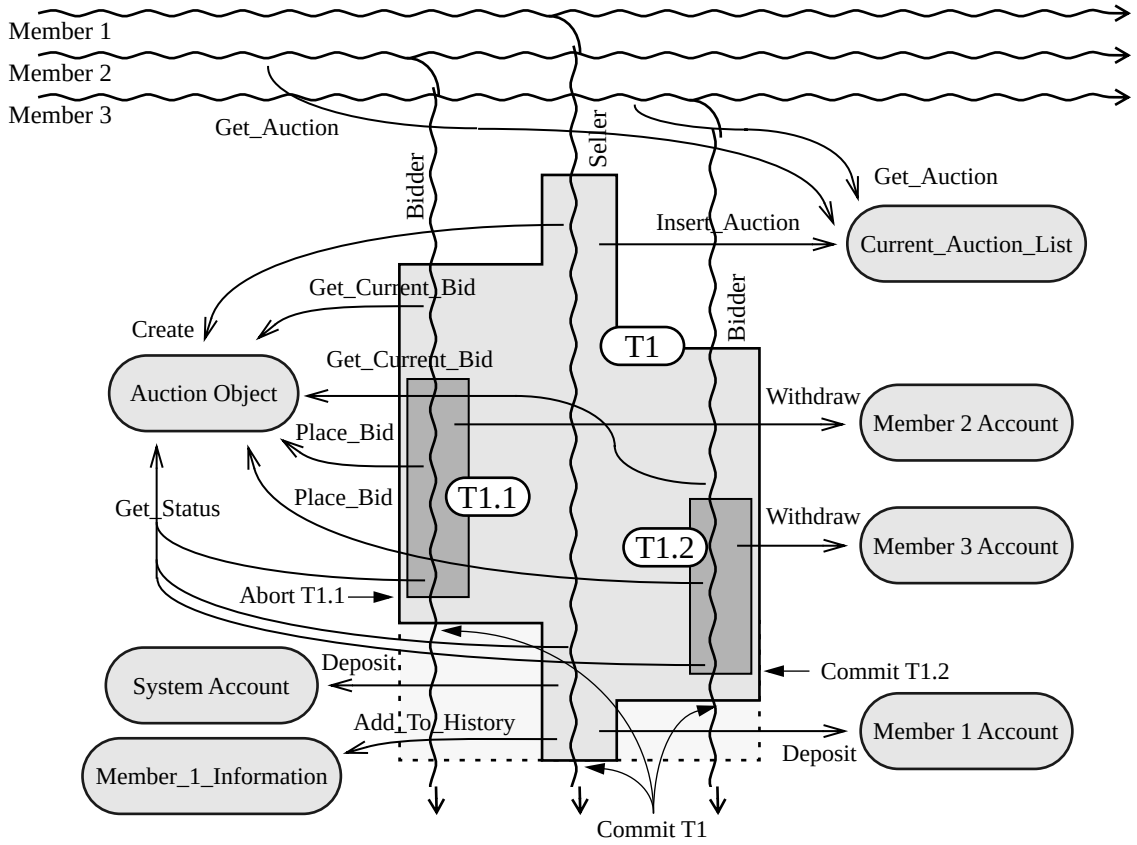

Figure 13.5: The *English Auction* Transaction

Other members consulting the current auction list will now see the new auction. In our example, member 2 decides to participate. A new bidder thread is created, which joins the open multithreaded transaction `T1`. It queries the amount of the current bid by invoking the `Get_Current_Bid` operation on the auction object. Before placing the bid, a new subtransaction, here named `T1.1`, is started. Within the subtransaction, the required amount of money is withdrawn from the account of member 2. Since there is enough money on the account, the withdrawal completes successfully and the bid is announced to the `Auction` object by calling `Place_Bid`.

In the meantime, some other member, member 3, joins the auction, spawning also a bidder thread, which joins the open multithreaded transaction `T1`. After consulting the current bid, member 3 decides to overbid member 2. Again, a subtransaction is started, here named `T1.2`, and the required amount of money is withdrawn from the account of member 3. The new bid is announced to the `Auction` object by calling `Place_Bid`. Once the bidder thread of member 2 gets to know this, it consequently aborts the subtransction `T1.1`, which in turn rolls back the withdrawal performed on the account of member 2. The money returned to the account of member 2 can now be used again for placing new bids.

In the example shown in figure 13.5, no other bidders enter the auction, nor does member 2 try to overbid member 3. The bidder thread of member 2 has therefore completed

its work inside the auction, and commits the global transaction `T1`. Once the auction closes, the bidder thread of member 3 gets to know that it has won the auction. It then commits the subtransaction `T1.2`, which confirms the previous withdrawal. It also commits the global transaction `T1`. The seller thread in the meantime deposits two percent of the amount of the final bid on the account of the auction system as a commission, deposits 98 % of the amount of the final bid on the account of member 1, inserts the `Auction` object into the history of the `Member_Information` object of member 1, and finally also commits `T1`.

Only now that all participants have voted commit, the transaction support will make the changes made on behalf of `T1` persistent, i.e. the creation of the auction object, the bidding, the withdrawal from the account of member 3 (inherited from subtransaction `T1.2`), the deposit on the auction system account, the deposit on the account of member 1, and the insertion of the auction object into the history of the `Member_Information` object of member 1.

The Ada implementation of the auction transaction is presented in section 13.3.3.2 on page 207.

13.3 Implementation

The last part of this chapter sketches the implementation of the auction system using the prototype implementation of the framework for Ada 95 presented in chapter 11.

Unfortunately, the OPTIMA framework currently does not support distribution. It is not possible to distribute the individual participants of an open multithreaded transaction onto different machines. For this reason, the functionality of the system is centralized, implemented on a single server. The program running on the client machines is just a simple user interface that forwards the individual requests to the server using the distribution features of Ada (section 10.5 on page 143).

13.3.1 Transactional Objects

This section illustrates how a conventional class can be transformed into a transactional class. As an example, the transformation of the `Account` class is presented, resulting in the creation of the `Transactional_Account` class.

13.3.1.1 `Account_Type` Implementation

The `Account` class has been implemented in the package `Accounts`, that defines the tagged type `Account_Type` together with the primitive operations `Deposit`, `Withdraw` and `Balance`. Figure 13.6 shows the specification and the body of the package `Accounts`.

```
with Ada.Streams; use Ada.Streams;
package Accounts is

   type Account_Type is
     limited private;

   type Account_Ref is
     access all Account_Type;

   procedure Deposit
     (Account : in out Account_Type;
      Amount  : in Natural);

   procedure Withdraw
     (Account : in out Account_Type;
      Amount  : in Natural);

   function Balance (Account :
     Account_Type) return Natural;

   Not_Enough_Funds : exception;

private

   type Account_Type is
     tagged limited record
     Balance : Natural := 0;
   end record;

end Accounts;
```

```
package body Accounts is

   procedure Deposit
     (Account : in out Account_Type;
      Amount: in Natural) is
   begin
     Account.Balance :=
       Account.Balance + Amount;
   end Deposit;

   procedure Withdraw
     (Account : in out Account_Type;
      Amount  : in Natural) is
   begin
     Account.Balance :=
       Account.Balance - Amount;
   exception
     when Constraint_Error =>
       raise Not_Enough_Funds;
   end Withdraw;

   function Balance (Account :
     Account_Type) return Natural is
   begin
     return Account.Balance;
   end Balance;

end Accounts;
```

Figure 13.6: Implementation of the `Accounts` package

13.3.1.2 `Transactional_Account_Type` Specification

In order to be able to take part in a transaction, the `Account` class must be transformed into a `Transactional_Account` class following the rules stated in section 9.2 on page 110.

The following paragraphs describe this transformation. Most of the work could be performed automatically by a pre-processor (see section 11.2.2 on page 160). The specification of the `Transactional_Account_Type`, for instance, can be derived from the specification of the `Account_Type` as shown in figure 13.7.

Three additional operations must be provided for transactional objects, namely `Create` ①, `Restore` ② and `Delete` ③. An object of type `Transactional_Account_Type` must store references to the associated memory object and atomic call object ④.

13.3.1.3 `Transactional_Account_Type` Implementation

The body of the `Transactional_Accounts` package is rather complicated, and therefore explained in incremental steps.

According to figure 9.4 on page 115, which shows all the classes that must be implemented to encapsulate a non-transactional data object, a transactional account must imple-

```
with Accounts; use Accounts;
with Atomic_Calls; use Atomic_Calls;
with Memory_Objects.Concrete; use Memory_Objects.Concrete;
with Storage_Params.Non_Volatile; use Storage_Params.Non_Volatile;

package Transactional_Accounts is

  type Transactional_Account_Type is limited private;

  type Transactional_Account_Ref is
    access all Transactional_Account_Type;

  procedure Deposit (Account : in out Transactional_Account_Type;
                     Amount  : in Natural);

  procedure Withdraw (Account : in out Transactional_Account_Type;
                      Amount  : in Natural);

  function Balance (Account : Transactional_Account_Type)
    return Natural;

  function Transactional_Balance (Account : Transactional_Account_Type)
    return Natural;

  function Create (Params : Non_Volatile_Params_Type'Class) (1)
    return Transactional_Account_Ref;

  function Restore (Params : Non_Volatile_Params_Type'Class) (2)
    return Transactional_Account_Ref;

  procedure Delete; (3)

  Not_Enough_Funds : exception renames Accounts.Not_Enough_Funds;

private

  type Transactional_Account_Type is limited record
    Atomic_Call : Atomic_Call_Ref;
    Memory_Object : Concrete_Memory_Object_Ref; (4)
  end record;

end Transactional_Accounts;
```

Figure 13.7: Specification of the `Transactional_Accounts` package

ment the classes `Account_Creation_Operation`, `Account_Loading_Operation`, `Account_Saving_Operation`, `Account_Deletion_Operation`, and, for each operation of the original data object, a class derived from `Normal_Operation` that encapsulates the operation invocation and the concurrency control information, encapsulated in a class derived from `Operation_Information`. The `Memory_Object` class, the `Atomic_Call` class, a concrete `Concurrency_Control` class and `Storage` class can be reused for all transactional objects.

13.3.1.4 Type Safety

Ada is a strongly typed language. The memory object, however, must be able to keep a reference to the data object, which can be of any type. In C++, for instance, a void pointer, i.e. a pointer that can point to anything, is used in such a situation. The more type-safe Ada

approach is to instantiate the generic procedure `Ada.Unchecked_Conversion`, creating a procedure that allows to convert an access type to some other access type. This technique is illustrated in figure 13.8. The type `Data_Ref` is defined as an access to a private type in the package `Memory_Objects`.

```
function To_Data_Ref is
  new Ada.Unchecked_Conversion (Account_Ref, Data_Ref);

function To_Account_Ref is
  new Ada.Unchecked_Conversion (Data_Ref, Account_Ref);
```

Figure 13.8: From `Account_Ref` to `Data_Ref`, and Vice Versa

13.3.1.5 Creation, Loading, Saving and Deletion

The memory object associated with a transactional account object must be able to create accounts, load accounts, save accounts and delete accounts. This functionality is encapsulated in the `Creation_Operation`, `Loading_Operation`, `Saving_Operation` and `Deletion_Operation` classes (see figure 9.2 on page 113). Concrete subclasses of these classes must be implemented for each transactional class. The implementation for the transactional account class is shown in figure 13.9.

The `Create` function of the `Account_Creation_Type` just allocates a new `Account_Type` object ①. The `Create_And_Load` function of the type `Account_Loading_Type` does alike, and subsequently initializes the state of the account from a `Stream` by using the predefined implementation of the `'Read` attribute ②. `Save` writes the state of the account to a `Stream` by means of the predefined attribute `'Write` ③, and `Delete` frees the memory associated with an account using an instantiation of `Ada.Unchecked_Deallocation` ④.

Based on these classes, the `Create` and `Restore` constructor functions are easy to implement. Figure 13.10 shows the implementation of the `Create` constructor function. It allocates a new transactional object ①, declares a creation, loading, saving and deletion object ②, and passes them to the constructor of the memory object class. The last two parameters specify that logical logging and in-place update will be used ③. Finally, an atomic call object is allocated ④.

13.3.1.6 Concurrency Control

Next, concurrency information for the operations `Deposit`, `Withdraw`, `Balance` and `Transactional_Balance` must be provided. This is illustrated in figure 13.11.
The `Account_Information_Type` derives from the abstract `Operation_Information_Type` (see section 7.4.3 on page 87). It encapsulates the concurrency control information for all four operations. The kind of operation is determined by the `Kind` discriminant ①.

```
type Account_Creation_Type is new Creation_Operation_Type
  with null record;

function Create (Operation : Account_Creation_Type) return Data_Ref is
  Result : Account_Ref := new Account_Type; (1)
begin
  return To_Data_Ref (Result);
end Create;

type Account_Loading_Type is new Loading_Operation_Type
  with null record;

procedure Create_And_Load (Operation   : Account_Loading_Type;
                           Data_Object : out Data_Ref;
                           Stream      : Abstract_Stream_Ref) is
  Result : Account_Ref := new Account_Type;
begin
  Account_Type'Read (Stream, Result.all); (2)
  Data_Object := To_Data_Ref (Result);
end Create_And_Load;

procedure Load (Operation   : Account_Loading_Type;
                Data_Object : Data_Ref;
                Stream      : Abstract_Stream_Ref) is
  Tmp : Account_Ref := To_Account_Ref (Data_Object);
begin
  Account_Type'Read (Stream, Tmp.all);
end Load;

type Account_Saving_Type is new Saving_Operation_Type with null record;

procedure Save (Operation   : Account_Saving_Type;
                Data_Object : Data_Ref;
                Stream      : Abstract_Stream_Ref) is
begin
  Account_Type'Write (Stream, To_Account_Ref (Data_Object).all); (3)
end Save;

type Account_Deletion_Type is new Deletion_Operation_Type
  with null record;

procedure Delete (Operation   : Account_Deletion_Type;
                  Data_Object : in out Data_Ref) is

  procedure Free_Account is new
    Ada.Unchecked_Deallocation (Account_Type'Class, Account_Ref);

  Tmp : Account_Ref := To_Account_Ref (Data_Object);
begin
  Free_Account (Tmp); (4)
  Data_Object := null;
end Delete;
```

Figure 13.9: Implementing Creation, Loading, Saving and Deletion

```
function Create (Params : Non_Volatile_Params_Type'Class)
  return Transactional_Account_Ref is                    (1)
  Result : Transactional_Account_Ref := new Transactional_Account_Type;
  My_Creation_Operation : Account_Creation_Type;
  My_Loading_Operation : Account_Loading_Type;
  My_Saving_Operation : Account_Saving_Type;
  My_Deletion_Operation : Account_Deletion_Type; (2)
begin
  Result.My_Memory_Object := Concrete_Memory_Object_Ref
    (Create (Params,
             My_Creation_Operation, My_Loading_Operation,
             My_Saving_Operation, My_Deletion_Operation,
             Logical, Inplace)); (3)
  Result.My_Atomic_Call :=
    new Atomic_Call_Type (Result.My_Memory_Object); (4)
  return Result;
end Create;
```

Figure 13.10: Implementing the `Create` Constructor for Transactional Accounts

```
type Operation_Kind_Type is
  (Deposit_Kind, Withdraw_Kind, Balance_Kind, Trans_Balance_Kind);

type Account_Information_Type (Kind : Kind_Type) is (1)
  new Operation_Information_Type with null record;

type Account_Information_Ref is
  access all Account_Information_Type'Class;

function Is_Compatible (Info : Account_Information_Type;
                        Other_Info : Account_Information_Type) (2)
  return Boolean is
begin
  case Info.Kind is
    when Deposit_Kind =>
      return not Other_Info.Kind = Balance_Kind;
    when Withdraw_Kind =>
      return Other_Info.Kind = Withdraw_Kind
      or Other_Info.Kind = Trans_Balance_Kind;
    when Balance_Kind =>
      return Other_Info.Kind = Balance_Kind
        or Other_Info.Kind = Trans_Balance_Kind;
    when Trans_Balance_Kind =>
      return True;
  end case;
end Is_Compatible;

function Is_Modifier (Info : Account_Information_Type) return Boolean is
begin (3)
  return Info.Kind = Deposit or Info.Kind = Withdraw;
end Is_Modifier;
```

Figure 13.11: Implementing Concurrency Control Information for Accounts

The operation Is_Compatible, required for inter-transaction concurrency control, implements the compatibility table presented in figure 13.3 ②. The operation Is_Modifier, required for intra-transaction concurrency control, classifies the Deposit and Withdraw operations as *modifiers* and the Balance and Transactional_Balance operations as *observers* ③.

13.3.1.7 Encapsulating Operations

Finally, the Deposit, Withdraw, Balance and Transactional_Balance operations must be encapsulated inside Normal_Operation objects (see section 9.2.3 on page 112).

Figure 13.12 shows how this is accomplished for the Deposit operation. First, Deposit_Operation_Type is derived from Normal_Operation_Type. The Deposit operation has one input parameter of type Natural, that represents the amount of money the caller wants to deposit on the account. Therefore, the Deposit_Operation_Type must declare a component Deposit_Amount of the same type, which can be used for storing the value of the input parameter Amount ①.

The Deposit operation of the transactional account declares an instance of the Deposit_Operation_Type, initializes the component Deposit_Amount with the value provided as an input parameter, and passes the declared instance on to the Atomic_Call object associated with the transactional object ②.

The Deposit_Operation_Type must implement four mandatory primitive operations, namely Do_Operation, Undo_Operation, Get_Operation_Info and Is_Update.

Do_Operation is invoked by the Memory_Object after the Atomic_Call object has performed the required concurrency control prologue and recovery prologue. After type-casting the Data_Object reference back to an Account_Ref reference and extracting the amount of money from the component Deposit_Amount, Do_Operation finally calls Deposit of the Account_Type ③.

The Undo_Operation is implemented in a similar way. Undoing a deposit comes down to withdrawing the amount of money that has been previously deposited ④.

Get_Operation_Info must return an operation information object encapsulating the concurrency control information for the Deposit operation. It therefore creates a new Account_Information_Type object with the appropriate discriminant, e.g. Deposit_Kind ⑤.

Since an invocation of Deposit modifies the durable state of an account object, Is_Update must return True ⑥.

13.3.2 Starting the System

Before starting any transaction, the programmer must initialize the transaction support. This is done by calling the procedure System_Init described in section 11.2.6 on page 169. In our implementation no parameters are supplied to the procedure, and hence the default implementation, i.e. LRU cache manager, a Redo/NoUndo recovery manager, and a mirrored file for storing the log are used.

```
type Deposit_Operation_Type is new Normal_Operation_Type with record
  Deposit_Amount : Natural; (1)
end record;

procedure Do_Operation (Operation   : in out Deposit_Operation_Type;
                        Data_Object : in Data_Ref) is
begin
  Deposit (To_Account_Ref (Data_Object).all,
           Operation.Deposit_Amount); (3)
end Do_Operation;

procedure Undo_Operation (Operation   : in out Deposit_Operation_Type;
                          Data_Object : in Data_Ref) is
begin
  Withdraw (To_Account_Ref (Data_Object).all,
            Operation.Deposit_Amount); (4)
end Undo_Operation;

function Get_Operation_Info (Operation : Deposit_Operation_Type)
  return Operation_Information_Ref is
begin
  return new Account_Information_Type (Deposit_Kind); (5)
end Get_Operation_Info;

function Is_Update (Operation : Deposit_Operation_Type)
  return Boolean is
begin
  return True; (6)
end Is_Update;

procedure Deposit (Account : in out Transactional_Account_Type;
                   Amount  : in Natural) is
  Deposit_Operation : Deposit_Operation_Type;
begin
  Deposit_Operation.Value := Value;
  Atomic_Do (Account.Atomic_Call.all, Deposit_Operation); (2)
end Deposit;
```

Figure 13.12: Encapsulating the `Deposit` Operation

13.3.3 Example Implementation of Open Multithreaded Transactions

This section presents the implementation of the *Registration* transaction and the *English auction* transaction of the auction system. The code uses the object-based interface for Ada presented in section 11.2.4 on page 163. For clarity, all code dealing with the graphical user interface has been omitted, and replaced by calls to the imaginary package `GUI`.

13.3.3.1 Registration

Figure 13.13 shows the implementation of the procedure that performs the registration of new members using a single-threaded, flat transaction conforming to the design presented in section 13.2.2.1 on page 195.

First, the user is asked to fill out the registration form. The code outlines this user interaction by calling operations of the package GUI ①. In window-based graphical user interfaces, this interaction might be implemented differently. The Member_Info_Params and Account_Params variables designate where the state of the new Account and Member_Information objects will be stored. In our example, they are stored on mirrored files. Mirrored files are declared in the package Storage.Non_Volatile.Stable.Mirrored_File_Storage and the corresponding storage parameters can be found in the package Storage.Non_Volatile.Stable.Mirrored_File_Storage_Params. The username of the new member, combined with the additional ending ".account" respectively ".member_info" are used as file names. The member directory is stored in a mirrored file named "member_directory" ②.

Next, a Transactional_Member_Info_Ref, a Transactional_Account_Ref and a Transactional_Member_Directory_Ref are declared. These references are needed to work with the transactional objects inside the transaction ③.

```
with Object_Based_Transaction_Interface;
use Object_Based_Transaction_Interface;

with Storage_Params.Non_Volatile.Stable.Mirrored_File_Storage_Params;
use Storage_Params.Non_Volatile.Stable.Mirrored_File_Storage_Params;

procedure Register_Client is
  Name : String := GUI.Get_Name;
  Password : String := GUI.Get_Password;
  Realname : String := GUI.Get_Realname;
  Address : String := GUI.Get_Address;
  Email : String := GUI.Get_Email;
  Initial_Deposit : Natural := GUI.Get_Initial_Deposit; (1)

  Account_Params : Mirrored_File_Storage_Params_Type :=
    String_To_Storage_Params (Name & ".account");
  Member_Info_Params : Mirrored_File_Storage_Params_Type :=
    String_To_Storage_Params (Name & ".member_info");
  Member_Directory_Params : Mirrored_File_Storage_Params_Type :=
    String_To_Storage_Params ("member_directory"); (2)

  New_Member : Transactional_Member_Info_Ref;
  New_Account : Transactional_Account_Ref;
  Member_Directory : Transactional_Member_Directory_Ref; (3)

  Register_Transaction : Transaction; (4)

begin
  New_Account := Create (Account_Params, Initial_Deposit); (5)
  New_Member := Create (Member_Info_Params, Username, Password, Realname,
                        Address, Email, Account_Params); (6)
  Member_Directory := Restore (Member_Directory_Params); (7)
  Register_Member (Member_Directory.all, Member_Params); (8)
  Commit_Transaction (Register_Transaction); (9)
end Register_Client;
```

Figure 13.13: Implementation of the *Registration* Transaction

Finally, the transaction is started by declaring the `Register_Transaction` object based on the `Transaction` type defined in the package `Object_Based_Transaction_Interface` ④. Inside the transaction, a new `Transactional_Account` is created by invoking the `Create` constructor function, passing as parameter the initial deposit. As a result, the mirrored file `"username.account"` is created on the local disk ⑤. Next, a `Transactional_Member_Information_Type` object is created, passing all relevant information about the new member as parameters to the `Create` constructor ⑥.

In order to register the member, the member directory must first be loaded. This is done by calling the `Restore` operation ⑦, which will read the directory from the mirrored file `"member_directory"`. Only then the member object can be registered in the directory by invoking `Register_Member` ⑧.

Last but not least, the transaction is committed by invoking `Commit_Transaction` on the `Register_Transaction` object ⑨. As a result, the creation and initialization of the account object, the creation and initialization of the member information object, and the changes made to the member directory are made durable.

13.3.3.2 English Auction Transaction

Figure 13.14 and figure 13.15 show the implementation of the `Seller_Task` and `Bidder_Task` that represent the seller and the bidders in an auction conforming to the design presented in section 13.2.2.1 on page 195.

Seller Implementation

Once a member has started a new auction and filled out the *item form*, a new seller task is created that represents the member in the auction. The information on the item form is transmitted to the task. This is symbolized in the code by calling operations of the package `GUI` ①. The `Auction_Params` and `List_Params` storage parameters identify where to store the stable state of the new `Auction` object, and where to retrieve the `Auction_List` object containing the list of current auctions ②. The references `Auction` and `Current_Auctions` are needed to manipulate the transactional objects ③.

Next, a named open multithreaded transaction encapsulating the auction is started. To name the transaction after the auction title, a reference to a string containing the title is passed as a discriminant constraint to the `Auction_Transaction` object ④. Then, a new `Transactional_Auction` object is created and initialized ⑤. The current auction list is retrieved ⑥, and the new auction object inserted into the list ⑦. Now, the seller task must wait until the auction closes ⑧, hoping that some bidders will join the auction and place bids.

Once the auction is closed, the seller task examines the current bid. If it is higher or equal to the opening bid, then at least one successful bid has been made ⑨. In that case, the account of the seller and the account of the auction system are loaded ⑩. 98% of the bid is

```
task body Seller_Task is

  Title : String := GUI.Get_Title;
  Description : String := GUI.Get_Description;
  Opening_Bid : Natural := GUI.Get_Opening_Bid;
  Timeout : Duration := GUI.Get_Timeout; (1)

  Auction_Params : Mirrored_File_Storage_Params_Type :=
    String_To_Storage_Params (Title & ".auction");
  List_Params : Mirrored_File_Storage_Params_Type :=
    String_To_Storage_Params ("auction_list"); (2)

  Auction : Transactional_Auction_Ref;
  Current_Auctions : Transactional_Auction_List_Ref; (3)
  Current_Bid : Natural;

  Auction_Transaction : Transaction := (new String' (Title), null, 0); (4)

begin

  Auction := Create (Auction_Params, Title, Description
                     Opening_Bid, Timeout); (5)
  Current_Auctions := Restore (List_Params); (6)
  Insert_Auction (Current_Auctions.all, Auction_Params); (7)

  while Get_Status (Auction.all) /= Closed loop
    delay A_While; (8)
  end loop;

  Current_Bid := Get_Current_Bid (Auction.all);
  if Current_Bid >= Opening_Bid then (9)
    declare
      Seller_Account : Transactional_Account_Ref :=
        Restore (String_To_Storage_Params (Username & ".account"));
      Auction_System_Account : Transactional_Account_Ref :=
        Restore (String_To_Storage_Params ("auction_system.account")); (10)
    begin
      Deposit (My_Account.all, Natural (Float (Current_Bid) * 0.98)); (11)
      Deposit (Auction_System_Account.all,
               Natural (Float (Current_Bid) * 0.02)); (12)
    end;
  end if;

  Commit_Transaction (Auction_Transaction); (13)
end Seller_Task;
```

Figure 13.14: Implementation of the *Seller* Task

transferred to the seller account (11), 2% of the bid is deposited on the auction system account (12). Finally, the transaction is committed by invoking the `Commit_Transaction` operation on the `Auction_Transaction` object (13).

Bidder Implementation

The implementation of the `Bidder_Task` is shown in figure 13.15. The `Bidder_Task` collaborates with the `Seller_Task` through the `Transactional_Auction` object.

```
task body Bidder_Task is

  Auction : Transactional_Auction_Ref := Members.Get_Auction;
  Username : String := Members.Get_Username;
  Title : String := Members.Get_Title; (1)

  My_Account : Transactional_Account_Ref
  Current_Bid : Natural;
  My_Bid : Natural; (2)

  Auction_Transaction : Transaction (new String' (Title), null, 0); (3)

begin
  My_Account :=
    Restore (String_To_Storage_Params (Username & ".account")); (2)

  while Get_Status (Auction.all) /= Closed loop (7)
    Current_Bid := Get_Current_Bid (Auction.all); (5)
    select (8a)
      GUI.Get_Bid_From_User (My_Bid, Current_Bid); (6)

      begin

        declare
          Subtransaction : Transaction; (9)
        begin
          Withdraw (My_Account.all, My_Bid); (10)
          Place_Bid (Auction.all, My_Bid); (12)

          while Bid_Accepted (Auction.all)
            and Get_Status (Auction.all) /= Closed loop
            delay A_While; (13)
          end loop;

          if Bid_Accepted (Auction.all) then
            Commit_Transaction (Subtransaction); (14)
          else
            GUI.Notify_User (Overbid);
            raise Transaction_Abort; (15)
          end if;

        exception
          when Not_Enough_Funds =>
            GUI.Notify_User (Not_Enough_Funds);
            raise Transaction_Abort; (11)
        end;

      exception
        when Transaction_Abort => null; (16)
      end;

    or
      delay A_While; (8b)
    end select;
  end loop;

  Commit_Transaction (Auction_Transaction); (17)
end Bidder_Task;
```

Figure 13.15: Implementation of the *Bidder* Task

When a member decides to participate in an auction, a new `Bidder_Task` is created. The reference to the `Auction` object representing the auction, the username of the member, and the title of the auction are handed to the `Bidder_Task`, e.g. by using an access discriminant, rendez-vous or protected object. In the code, this is symbolized using calls to the package `Members` ①. Next, an `Transactional_Account_Ref` used for manipulating the member account, and other auxiliary variables are declared ②. The `Bidder_Task` then joins the open multithreaded transaction started by the `Seller_Task` by also declaring a transaction object `Auction_Transaction`, passing the same name, i.e. the title of the auction, as discriminant constraint ③.

Inside the transaction, the member account is loaded ④. The `Bidder_Task` now obtains the current bid from the `Auction` object ⑤, and asks the member to place a bid by calling an operation of the package `GUI` ⑥. As long as the auction is not closed, the member is allowed to place a bid ⑦. If the user does not place a bid, the current bid is updated periodically using a timed select statement ⑧a ⑧b.

If the user places a bid, a new subtransaction is started by opening a new Ada block and declaring another transaction object ⑨. An attempt is made to withdraw the required money from the member's bank account ⑩. If there is not enough money on the account, the account object raises the `Not_Enough_Funds` exception. The internal exception is handled locally: a notification is sent to the user, and the subtransaction is aborted ⑪. If the withdrawal succeeds, the bid is sent to the `Auction` object ⑫.

Now, the `Bidder_Task` must wait until either the auction closes, or some other bidder places a higher bid ⑬. If finally the bid is accepted, the subtransaction is committed ⑭. Otherwise, the user is notified that someone else has placed a higher bid, the subtransaction is aborted, resulting in a rollback of the `Withdraw` operation, and the user is given a chance to place a new bid ⑮.

Each time a member overbids some other member, a subtransaction is aborted. Attempts to overdraw a member account also result in aborting a subtransaction. Such rollbacks are part of the normal life cycle of an auction, and should not affect the outcome of the auction in general. This is why the external exception `Transaction_Abort` propagated by the subtransaction upon rollback can be safely ignored ⑯.

In any case, once the auction closes, the global transaction is committed by invoking the `Commit_Transaction` operation on the `Auction_Transaction` object ⑰.

Additional Considerations

When there is not enough money on the account, or when some other member places a higher bid, the `Bidder_Task` aborts the subtransaction by raising the external exception `Transaction_Abort` ⑪ ⑮. It is good programming style to explicitly raise this external exception, but it is not necessary. The implementation could rely on the fact that the object-

based interface detects deserters, i.e. tasks that forget to vote on the outcome of a transaction. Hence, the transaction support would abort the transaction anyway.

The `Seller_Task` and the `Bidder_Task` use polling to coordinate their work inside the auction. A more elegant solution would be to suspend the tasks and wake them up if anything significant happens.

This can be achieved by adding synchronization to the operations of the `Transactional_Auction_Type` class. After executing a `Place_Bid` operation on the encapsulated `Auction` object, the `Transactional_Auction` object could suspend the calling task until some other task calls `Place_Bid` or the auction expires[1].

Another solution to this problem is to add a new operation `Get_Coordinator_Object` to the `Auction` object. The operation returns a protected type that all participants of the auction subsequently use to synchronize.

1. The synchronization can not be added to the encapsulated Auction class, because the mutual exclusion lock acquired in the concurrency control prologue (see section 9.2.3 on page 112) must first be released.

Chapter 14:
Conclusion

14.1 Summary of Results

This thesis has tackled the problems that arise when integrating transactions and concurrent object-oriented programming. In particular, it investigated the problems that arise when providing transaction support for modern programming languages, i.e. programming languages with support for concurrency and distribution.

In order to support *cooperative* and *competitive* concurrency, the classic single-threaded transaction model must be extended. An ideal model must allow multiple threads to be associated with the same transaction context, and still enforce the ACID properties. An analysis of existing transaction models has shown that they either give too much freedom to threads and do not control their participation in transactions, or unnecessarily restrict the computational model by assuming that only one thread can enter a transaction.

For this reason, a new transaction model named *Open Multithreaded Transactions* has been introduced, providing features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a common activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior when necessary in order to guarantee correctness of transaction nesting and enforcement of the ACID properties.

The open multithreaded transaction model incorporates disciplined exception handling, well adapted to nested multithreaded transactions. A clear distinction is made

between internal exceptions and external exceptions. Internal exceptions allow individual threads to perform *forward error recovery* by handling an abnormal situation locally. The open multithreaded transaction model promotes a defensive approach for developing transactional objects, so that errors are detected early and dealt with inside the transaction. Self-checking transactional objects may raise an exception if their consistency criteria is violated. If the participant that invoked the operation can not handle the exception locally, the transaction support automatically reverses the system to a previous consistent state by aborting the transaction. Thus, if forward error recovery is unsuccessful, open multithreaded transactions provide automatic *backward error recovery*. With this behavior, open multithreaded transactions act as firewalls for errors and hence constitute units of fault tolerance.

Transactions in general require considerable run-time support. The detailed design of the OPTIMA framework, a framework providing support for open multithreaded transactions, has been laid out. It offers transaction control, concurrency control, recovery support and persistence support. In order to support applications from many different domains, the framework provides means to customize and tailor it to specific application requirements. In order to maximize modularity and flexibility, the framework makes heavy use of design patterns.

To integrate open multithreaded transactions with a concurrent object-oriented programming language, a procedural, an object-based, and an object-oriented interface for the framework have been defined. These interfaces are based on standard programming techniques only, and do not use any “magic”.

To demonstrate the feasibility and usefulness of the OPTIMA framework it has been implemented for the concurrent object-oriented programming language Ada 95. The implementation takes the form of a library, and is based on standard Ada only. As a consequence, it is usable on any settings and platforms that have standard Ada compilers. The implementation has demonstrated the soundness of the OPTIMA approach. Having a working implementation of open multithreaded transactions allows application programmers to experiment with the model, and also to evaluate the advantages and disadvantages of the procedural, object-based and object-oriented interfaces.

Finally, the implementation made it possible to conduct a realistic case study. The auction system, an example of a dynamic system with cooperative and competitive concurrency, shows how the inherent complexity of the application can be reduced by structuring the execution with open multithreaded transactions. Reasoning about fault tolerance issues and consistency of the overall system is also made a lot easier. Due to the isolation property and disciplined exception handling, open multithreaded transactions do not allow errors to propagate to the outside, and therefore constitute units of fault tolerance.

14.2 Future Work

14.2.1 Extending the OPTIMA Framework to Support Distribution

One direction of future work is clearly concerned with adding support for distribution to the OPTIMA framework. In a distributed setting, different fault assumptions must be applied. In particular, the individual components of the system may fail separately.

Two main problems must be solved, namely providing *distributed access to transactional objects* and *distributed transaction control*.

14.2.1.1 Distributed Access to Transactional Objects

In the current framework, the states of all transactional objects are located on storage units that can be accessed by the computer on which the application using the framework is running. When an operation is invoked on a transactional object, the cache manager loads the state of the object from the associated storage unit into memory.

In a distributed setting, the individual participants of an open multithreaded transaction may execute on different computers, but must still be able to call operations of the same transactional object.

Different solutions are conceivable, somehow similar to the different alternatives that an EJB container provider faces when having to deal with concurrent access to the same EJB from multiple transactions [SHM+00, EJB.9.1.11].

Maybe the best solution is to have only one instance of the transactional object at some node of the distributed system, and forward all operation invocations made by participants executing on different nodes to this node[1]. In that case, the concurrency control of the transactional object remains the same. The cache manager, however, must be extended to keep track of the node on which a transactional object is located. This means that all cache managers of the distributed system must be somehow synchronized. If a local request is made to instantiate a transactional object that has already been instantiated on some remote node, then the cache manager must create a stub object instead that forwards all operation invocations to the real transactional object on the remote machine.

Another solution is to instantiate the transactional object on all machines that want to work with it. In that case, of course, the state and concurrency control of all these “replica” objects must be synchronized, which requires very time consuming communication protocols.

1. This is readily possible, since all operation invocations are anyway encapsulated in serializable subclasses of the `Normal_Operation` class.

14.2.1.2 Distributed Transaction Control

Since participants of an open multithreaded transaction potentially execute on different nodes of a distributed system, the transaction support component presented in chapter 6 must be extended to support distribution. Each participant sends its vote to the local transaction support component. Once all local votes have been collected, the local transaction support component must communicate with all other nodes involved in the transaction, since the outcome of the transaction can only be determined when the votes of all participants have been collected.

Moreover, because transactional objects used in an open multithreaded transaction are potentially located on different nodes, the recovery manager must be extended to perform a two-phase commit protocol among the transactional objects accessed from inside the transaction.

Finally, implementations of the distributed framework must also extend the communication mechanism used to connect the nodes of the distributed system. If a participant of an open multithreaded transaction makes a remote procedure call to some other node, the remote procedure must also be executed on behalf of the transaction. Hence, the transaction context must transparently be sent to the remote node.

14.2.2 Interacting with the CORBA Object Transaction Service

Although it is possible and it makes sense to add distribution to the framework itself, it might be more realistic to integrate open multithreaded transactions with existing transactional middlewares that provide communication for large-scale distributed systems. The CORBA standard provides distributed objects, and a naming service that makes it possible to determine on which node an object resides. The CORBA Object Transaction Service (see section 12.8 on page 181) provides basic transaction support, and performs the two-phase commit protocol between objects participating in the same transaction, so-called *resources*. The CORBA model of transactions allows multithreading inside a transaction, but leaves thread coordination inside a transaction to the application programmer.

Implementing open multithreaded transactions on top of the functionality offered by the CORBA Object Transaction Service could overcome this drawback. The `Synchronization` interface provided for resources, or other CORBA services, such as the CORBA Concurrency Service, might be useful for achieving thread control in a distributed CORBA environment. Probably, some sort of controller object that implements the open multithreaded transactions protocol must be designed. It would have to provide methods that allow individual participants to start, join, commit and abort open multithreaded transactions. The individual participants of an open multithreaded transaction would interact with the controller object, and the controller object would interact with the CORBA OTS.

Another promising approach is to use the OPTIMA framework to implement CORBA resources. The CORBA OTS does not provide automatic support for recoverability or per-

sistence. Resources must implement this functionality on their own, by forwarding the responsibility to an underlying database system, or by using other CORBA services. In all three cases, the task is rather complicated. The OPTIMA framework, on the other hand, offers everything that is needed to implement a CORBA resource, e.g. concurrency control, recoverability and persistence. By designing and implementing a bridge between the CORBA resource interface and the OPTIMA framework, the CORBA OTS can directly communicate with the transaction support component and recovery component of the framework, and the two-phase commit protocol can be performed automatically, without further help from the application programmer.

14.2.3 Formalizing the Open Multithreaded Transaction Model

The ACTA framework [CR90] is a first-order logic-based formalism that allows a transaction modeler to specify the behavioral properties of transaction models. ACTA characterizes the semantics of transactions in terms of effects that a transaction has on transactional objects and on other transactions. The effects on objects are expressed in terms of visibility of state, conflicts between operations, and delegation rules. The effects on other transactions are expressed in terms of inter-transaction dependencies, e.g. commit and abort dependencies.

ACTA has been used to formalize many existing transaction models, and even to synthesize new ones [CR94]. Formalizing the open multithreaded transaction model using ACTA would allow to verify the consistency of the open multithreaded transaction rules, provide a basis for proving relevant properties, and make it a lot easier to compare the model with other transaction models that have already been formalized.

14.2.4 Experimenting with Aspect-Oriented Programming Techniques

Section 9.5 on page 123 discusses the possibility of using reflection or aspect-oriented programming techniques to integrate the OPTIMA framework with a programming language. Aspect-oriented programming has particularly captured my attention, and I therefore intend to investigate how aspects can be used to provide elegant interfaces to transactions, and if aspects can be used to transform conventional objects into transactional objects.

The only aspect-oriented programming environment currently available is AspectJ [KHH+01], a programming environment for the Java programming language. I intend to conduct my experiments using this environment. A prototype implementation of the OPTIMA framework for the Java language is already under construction.

Part V

Annexes

Annex A: Bibliography

[ABC+83] Atkinson, M. P.; Bailey, P. J.; Chisholm, K. J.; Cockshott, W. P.; Morrison, R.: “An Approach to Persistent Programming”, *Computer Journal **26**(4)*, 1983, pp. 360 – 365.

[ACC81] Atkinson, M. P.; Chisholm, K. J.; Cockshott, W. P.: “PS-Algol: An Algol with a Persistent Heap”, *ACM SIGPLAN Notices **17**(7)*, July 1981, pp. 24 – 31.

[ACF99] Ancona, M.; Cazzola, W.; Fernandez, E. B.: “Reflective Authorization Systems: Possibilities, Benefits, and Drawbacks”, in *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pp. 35 – 50, Lecture Notes in Computer Science **1603**, Springer Verlag, 1999.

[ADJ+96] Atkinson, M. P.; Daynès, L.; Jordan, M. J.; Printezis, T.; Spence, S.: “An orthogonally persistent Java”, *ACM SIGMOD Record **25**(4)*, December 1996, pp. 68 – 75.

[ADS96] Atkinson, M. P.; Daynès, L.; Spence, S.: “Draft PJava Design 1.2”. *Technical report*, Department of Computing Science, University of Glasgow, January 1996.

[AJDS96] Atkinson, M. P.; Jordan, M. J.; Daynès, L.; Spence, S.: “Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System”, in *Proceedings of the 6th International Workshop on Persistent Object Systems*, Cape May, NJ, USA, May 1996.

[AM95] Atkinson, M. P.; Morrison, R.: “Orthogonally Persistent Object Systems”, *The VLDB Journal **4**(3)*, 1995, pp. 319 – 401.

[Arj00] Arjuna Solutions Limited: *JTSArjuna 2.0, Programmer’s Guide Volume 3*, 2000.

[AS89] Agrawal, D.; Sengupta, S.: “Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control”, in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*,

Portland, Oregon, May 31 - June 2, 1989, pp. 408 – 417, New York, USA, June 1989, ACM Press.

[Bad79] Badal, D. Z.: "Correctness of Concurrency Control and Implications for Distributed Databases", in *Proceedings of the IEEE International Computer Software and Application Conference – COMPSAC 79*, Chicago, USA, November 1979.

[Bar97] Barnes, J. (Ed.): *Ada 95 Rationale*. Lecture Notes in Computer Science **1247**, Springer Verlag, 1997.

[BCF+97] Besancenot, J.; Cart, M.; Ferrié, J.; Guerraoui, R.; Pucheral, P.; Traverson, B.: *Les Systèmes Transactionnels: Concepts, Normes et Produits*. Editions Hermes, Paris, France, 1997.

[Bes96] Best, E.: *Semantics of Sequential and Parallel Programs*. Prentice Hall, New York, NY, 1996.

[BG81] Bernstein, P. A.; Goodman, N.: "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys **13**(2)*, June 1981, pp. 185 – 221.

[BGL98] Briot, J.-P.; Guerraoui, R.; Lohr, K.-P.: "Concurrency and Distribution in Object-Oriented Programming", *ACM Computing Surveys **30**(3)*, September 1998, pp. 291 – 329.

[BH73] Brinch Hansen, P.: *Operating System Principles*. Prentice Hall, 1973.

[BHG87] Bernstein, P. A.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BI92] Banatre, J. P.; Issarny, V.: "Exception Handling in Communicating Sequential Processes". *Technical Report 660*, Inria, Institut National de Recherche en Informatique et en Automatique, June 1992.

[Bir85] Birman, K. P.: "Replication and Fault-Tolerance in the ISIS System", *ACM Operating Systems Review **19**(5)*, 1985, pp. 79 – 86.

[BN84] Birrell, A. D.; Nelson, B. J.: "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems **2**(1)*, 1984, pp. 39 – 59.

[Boo91] Booch, G.: *Object-Oriented Design with Applications*. Benjamin/Cummings Series in Ada and Software Engineering, The Benjamin/Cummings Publishing Company, Inc., Redwood City , CA , USA, 1991.

[Boo94] Booch, G.: *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 2nd ed., 1994.

[BP95] Barga, R.; Pu, C.: “A Practical and Modular Method to Implement Extended Transaction Models”, in *Proceedings of the 21st International Conference on Very Large Data Bases, Zürich, Switzerland, Sept. 11-15, 1995*, pp. 206 – 217, Los Altos, CA, USA, 1995, Morgan Kaufmann.

[BvR94] Birman, K. P.; van Renesse, R. (Eds.): *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[BW95] Burns, A.; Wellings, A. J.: *Concurrency in Ada*. Cambridge University Press, 1995.

[CAB+94] Coleman, D.; Arnold, P.; Bodoff, S.; Dollin, C.; Gilchrist, H.; Hayes, F.; Jeremaes, P.: *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Englewood Cliffs, 1994.

[CKS01] Caron, X.; Kienzle, J.; Strohmeier, A.: “Object-Oriented Stable Storage based on Mirroring”, in *Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001*, pp. 278 – 289, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.

[CR86] Campbell, R. H.; Randell, B.: “Error Recovery in Asynchronous Systems”, *IEEE Transactions on Software Engineering **SE-12**(8)*, August 1986, pp. 811 – 826.

[CR90] Chrysanthis, P. K.; Ramamritham, K.: “ACTA. A Framework for Specifying and Reasoning about Transaction Structure and Behavior”, *SIGMOD Record (ACM Special Interest Group on Management of Data) **19**(2)*, June 1990, pp. 194 – 203.

[CR94] Chrysanthis, P. K.; Ramamritham, K.: “Synthesis of Extended Transaction Models Using ACTA”, *ACM Transactions on Database Systems **19**(3)*, September 1994, pp. 450 – 491.

[Cri91] Cristian, F.: “Understanding Fault–Tolerant Distributed Systems”, *Communications of the ACM **34**(2)*, February 1991, pp. 56 – 78.

[Cri95] Cristian, F.: *Exception Handling and Tolerance of Software Faults*, pp. 81 – 108. in Lyu [Lyu95], 1995.

[CS95] Coplien, J. O.; Schmidt, D. C. (Eds.): *Pattern Languages of Program Design*. Addison–Wesley, Reading , MA , USA, 1995.

[Day96] Daynès, L.: “Extensible Transaction Management in PJava”, in *Proceedings of the First International Workshop on Persistence and Java, University of Glasgow, UK*, September 1996.

[DPSW89] Dixon, G. N.; Parrington, G. D.; Shrivastava, S. K.; Wheater, S. M.: "The Treatment of Persistent Objects in Arjuna", in *3rd European Conference on Object–Oriented Programming (ECOOP '89)*, pp. 169 – 189, Nottingham, July 1989, Cambridge University Press.

[DRJLAR91] Dasgupta, P.; Richard J. LeBlanc, J.; Ahamad, M.; Ramachandran, U.: "The Clouds Distributed Operating System", *Computer **24**(11)*, November 1991, pp. 34 – 44.

[EGLT76] Eswaran, K. P.; Gray, J.; Lorie, R. A.; Traiger, I. L.: "The Notion of Consistency and Predicate Locks in a Database System", *Communications of the ACM **19**(11)*, November 1976, pp. 624 – 633.

[Elm93] Elmagarmid, A. K. (Ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1993.

[EMS91] Eppinger, J. L.; Mummert, L. B.; Spector, A. Z.: *Camelot and Avalon - A Distributed Transaction Facility.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[FLW92] Fekete, A.; Lynch, N.; Weihl, W. E.: "Hybrid Atomicity for Nested Transactions", in *Proceedings of the 4th International Conference on Database Theory (ICDT'92)*, pp. 216 – 230, Berlin, Germany, October 1992, Lecture Notes in Computer Science **646**, Springer Verlag.

[GBCvR93] Glade, B. B.; Birman, K. P.; Cooper, R. C.; van Renesse, R.: "Light-weight Process Groups in the ISIS System", *Distributed Systems Engineering **1**(1)*, 1993, pp. 29 – 36.

[GGM94] Garbinato, B.; Guerraoui, R.; Mazouni, K. R.: "Distributed Programming in GARF", in Guerraoui, R.; Nierstrasz, O.; Riveill, M. (Eds.), *Object-Based Distributed Programming*, Lecture Notes in Computer Science **791**, pp. 225 – 239, Springer Verlag, Berlin, 1994.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[GJS96] Gosling, J.; Joy, B.; Steele, G. L.: *The Java Language Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 1996.

[GMAA97] Guerra, F.; Miranda, J.; Alvarez, A.; Arévalo, S.: "An Ada Library to Program Fault-Tolerant Distributed Applications", in *Proceedings of Reliable Software Technologies – Ada–Europe '97*, pp. 230 – 243, London, UK, June 1997, Lecture Notes in Computer Science **1251**, Springer Verlag.

[GMS87] Garcia-Molina, H.; Salem, K.: “SAGAS”, in *Proceedings of the SIGMod 1987 Annual Conference*, pp. 249 – 259, San Francisco, Ca, May 1987, ACM Press.

[Goo75] Goodenough, J. B.: “Exception Handling: Issues and a Proposed Notation”, *Communications of the ACM **18**(12)*, December 1975, pp. 683 – 696.

[GR93] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Han73] Hansen, P. B.: *Operating System Principles*. Prentice Hall, Englewood Cliffs, 1973.

[HCL90] Haritsa, J. R.; Carey, M. J.; Livny, M.: “On Being Optimistic about Real-Time Constraints”, in *PODS ’90. Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: April 2 - 4, 1990, Nashville, Tennessee*, volume 51 (1) of *Journal of Computer and Systems Sciences*, pp. 331 – 343, New York, NY 10036, USA, 1990, ACM Press.

[HKM+94] Haines, N.; Kindred, D.; Morrisett, J. G.; Nettles, S. M.; Wing, J. M.: “Composing First-Class Transactions”, *ACM Transactions on Programming Languages and Systems **16**(6)*, Nov 1994, pp. 1719 – 1736.

[Hoa74] Hoare, C. A. R.: “Monitors: An Operating Systems Structuring Concept”, *Communications of the ACM **17**(10)*, October 1974, pp. 549 – 557, ACM.

[Hoa75] Hoare, C. A. R.: “Parallel Programming: an Axiomatic Approach”, in *Proceedings of the International Summer School on Language Hierarchies and Interfaces*, pp. 11 – 42, Marktoberdorf, Germany, July 1975, Lecture Notes in Computer Science **46**, Springer Verlag.

[HR73] Horning, J. J.; Randell, B.: “Process Structuring”, *ACM Computing Surveys **5**(1)*, March 1973, pp. 5 – 30.

[ISO95] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.

[ISO99] ISO: *International Standard ISO/IEC 15291:1999 (E): Ada Semantic Interface Specification (ASIS)*, ISO, 1999.

[Iss93] Issarny, V.: “An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards the Design of Reusable, Robust Software”, *Journal of Object-Oriented Programming **6**(6)*, 1993, pp. 29 – 40.

[IY91] Ichisugi, Y.; Yonezawa, A.: “Exception Handling and Real-Time Features in Object-Oriented Concurrent Languages”, in *Concurrency: Theory, Language and Architecture*, pp. 92 – 109, Lecture Notes in Computer Science **491**, Springer Verlag, 1991.

[JPPMA00] Jiménez-Peris, R.; Patiño-Martinez, M.; Arévalo, S.: “TransLib: An Ada 95 Object-Oriented Framework for Building Transactional Applications”, *Computer Systems: Science & Engineering Journal **15**(1)*, 2000, pp. 7 – 18.

[KdB91] Kiczales, G.; des Rivieres, J.; Bobrow, D. G.: *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[KHH+01] Kiczales, G.; Hilsdale, E.; Hungunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: “An Overview of AspectJ”, in *15th European Conference on Object-Oriented Programming, ECOOP 2001*, Lecture Notes in Computer Science, 2001. To be published.

[Kie97] Kienzle, J.: “Network Applications in Ada 95”, in *TRI-Ada’97 Conference*, pp. 3 – 9, St. Louis, MO, November 1997, ACM Press.

[Kie99] Kienzle, J.: “Combining Tasking and Transactions”, in *Proceedings of the 9th International Real-Time Ada Workshop, Wakulla Springs Lodge, Tallahassee FL, USA, March 1999*, pp. 49 – 53, Ada Letters **XIX(2)**, ACM Press, June 1999.

[Kie00] Kienzle, J.: “Exception Handling in Open Multithreaded Transactions”, in *ECOOP Workshop on Exception Handling in Object-Oriented Systems, Cannes, France*, June 2000.

[KJPRPM01] Kienzle, J.; Jiménez-Peris, R.; Romanovsky, A.; Patiño-Martinez, M.: “Transaction Support for Ada”, in *Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001*, pp. 290 – 304, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: “Aspect-Oriented Programming”, in *11th European Conference on Object–Oriented Programming (ECOOP ’97)*, pp. 220 – 242, Jyváskylá, Finland, 1997, Lecture Notes in Computer Science **1241**, Springer Verlag.

[Knu87] Knudsen, J. L.: “Better Exception-Handling in Block-Structured Systems”, *IEEE Software **4**(3)*, May 1987, pp. 40 – 49.

[Kop97] Kopetz, H.: *Real–Time Systems — Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[KR81] Kung, H. T.; Robinson, J. T.: “On Optimistic Methods for Concurrency Control”, *ACM Transactions on Database Systems* ***6****(2)*, June 1981, pp. 213 – 226.

[KR00] Kienzle, J.; Romanovsky, A.: “On Persistent and Reliable Streaming in Ada”, in *Reliable Software Technologies - Ada-Europe’2000, Potsdam, Germany, June 26-30, 2000*, pp. 82 – 95, Lecture Notes in Computer Science **1845**, 2000.

[KR01] Kienzle, J.; Romanovsky, A.: “Combining Tasking and Transactions, Part II: Open Multithreaded Transactions”, in *Proceedings of the 10th International Real-Time Ada Workshop, Castillo de Magalia, Las Navas del Marqués, Avila, Spain, September 2000*, pp. 67 – 74, Ada Letters **XXI(1)**, ACM Press, March 2001.

[KRS00] Kienzle, J.; Romanovsky, A.; Strohmeier, A.: “A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages”. *Technical Report EPFL-DI No 2000/335*, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

[KRS01] Kienzle, J.; Romanovsky, A.; Strohmeier, A.: “Open Multithreaded Transactions: Keeping Threads and Exceptions under Control”, in *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Roma, Italy, 8 - 10 January, 2001*, 2001. To be published.

[KS99] Kienzle, J.; Strohmeier, A.: “Shared Recoverable Objects”, in *Reliable Software Technologies - Ada-Europe’99, Santander, Spain, June 7-11, 1999*, volume 1622 of *Lecture Notes in Computer Science*, pp. 397 – 411, 1999.

[KSM98] Kurki-Suonio, R.; Mikkonen, T.: “Liberating Object-Oriented Modeling from Programming-Level Abstractions”, *Lecture Notes in Computer Science* ***1357***, 1998, pp. 195 – 199.

[KWS96] Kienzle, J.; Wolf, T.; Strohmeier, A.: “Secure Communication in Distributed Ada”, in *Reliable Software Technologies - Ada-Europe’96, Montreux, Switzerland, June 10-14, 1996*, pp. 198 – 210, Lecture Notes in Computer Science **1088**, Springer Verlag, 1996.

[LA90] Lee, P. A.; Anderson, T.: “Fault Tolerance - Principles and Practice”, in *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2nd ed., 1990.

[LAB+81] Liskov, B.; Atkinson, R.; Bloom, T.; Moss, J. E. B.; Schaffert, J. C.; Scheifler, R.; Snyder, A.: *CLU Reference Manual*. Lecture Notes in Computer Science **114**, Springer Verlag, 1981.

[Lap85] Laprie, J.-C.: "Dependable Computing and Fault Tolerance : Concepts and Terminology", in *Proceedings of the 15th International Symposium on Fault–Tolerant Computing Systems (FTCS–15)*, pp. 2 – 11, Ann Arbour, MI, USA, June 1985.

[Lis85] Liskov, B.: "The Argus Language and System", in *Distributed Systems, Methods and Tools for Specification*, pp. 343 – 430, Lecture Notes in Computer Science **190**, Springer-Verlag, 1985.

[Lis87] Liskov, B.: "Implementation of Argus", in *Proceedings of the 11th ACM Symposium on Operating Systems Principle*, pp. 111 – 122, November 1987.

[Lis88] Liskov, B.: "Distributed Programming in Argus", *Communications of the ACM **31**(3)*, March 1988, pp. 300 – 312.

[LJP93] Lea, R.; Jacquemot, C.; Pillevesse, E.: "COOL: System Support for Distributed Programming", *Communications of the ACM **36**(9)*, September 1993, pp. 37 – 46.

[LS79] Lampson, B. W.; Sturgis, H. E.: "Crash Recovery in a Distributed Data Storage System". *Technical report*, XEROX Research, Palo Alto, June 1979.

[LS83] Liskov, B.; Scheifler, R.: "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems **5**(3)*, July 1983, pp. 381 – 404.

[LSP82] Lamport, L.; Shostak, R.; Pease, M.: "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems **4**(3)*, 1982, pp. 382 – 401.

[Lyu95] Lyu, M. R. (Ed.): *Software Fault Tolerance*. John Wiley & Sons, 1995.

[LZ94] Lee, A. H.; Zachary, J. L.: "Using Metaprogramming to Add Persistence to CLOS", in *Proceedings of the 1994 International Conference on Computer Languages, May 16–19, 1994, Toulouse, France*, pp. 136 – 147, Silver Spring, MD, USA, 1994, IEEE Computer Society Press.

[MAAG96] Miranda, J.; Alvarez, A.; Arévalo, S.; Guerra, F.: "Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications", in *Reliable Software Technologies - Ada-Europe'96, Montreux, Switzerland, June 10-14, 1996*,

pp. 235 – 246, Lecture Notes in Computer Science **1088**, Springer Verlag, 1996.

[Mae87] Maes, P.: "Concepts and Experiments in Computational Reflection", *ACM SIGPLAN Notices **22**(12)*, December 1987, pp. 147 – 155.

[Mat87] Matthews, D. C. J.: "A persistent storage system for Poly and ML". *Technical Report TR-102*, Computer Laboratory, University of Cambridge, January 1987.

[Mey88] Meyer, B.: *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, Prentice Hall International, Hemel Hempstead, UK, 1988.

[Mey92] Meyer, B.: *Eiffel: The Language*. Object-Oriented Series, Prentice Hall, New York, NY, 1992.

[Mey97] Meyer, B.: *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ 07632, USA, 2nd ed., 1997.

[MMPN93] Madsen, O. L.; Møller-Pedersen, B.; Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.

[MN82] Menascé, D. A.; Nakanishi, T.: "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", *Information Systems **7**(1)*, 1982, pp. 13 – 27.

[Mos81] Moss, J. E. B.: *Nested Transactions, An Approach to Reliable Computing*. PhD Thesis, MIT, Cambridge, April 1981.

[MRB98] Martin, R.; Riehle, D.; Buschmann, F. (Eds.): *Pattern Languages of Program Design 3*. Addison–Wesley, Reading, MA, USA, 1998.

[MW91] Mössenböck, H.; Wirth, N.: "The Programming Language Oberon–2". *Technical Report 160*, Institut für Computersysteme, Swiss Federal Institute of Technology, Zürich, Switzerland, May 1991.

[MY93] Matsuoka, S.; Yonezawa, A.: "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", in Agha, G.; Wegner, P.; Yonezawa, A. (Eds.), *Research Directions in Concurrent Object-Oriented Programming*, pp. 107 – 150, MIT Press, 1993.

[NP90] Nierstrasz, O.; Papathomas, M.: "Viewing Objects as Patterns of Communicating Agents", *Proceedings of OOPSLA / ECOOP'90, ACM SIGPLAN Notices **25**(10)*, October 1990, pp. 38 – 43.

[Obj95] Object Management Group, Inc.: *The Common Object Request Broker Architecture Specification, Revision 2.0*, 1995.

[Obj00] Object Management Group, Inc.: *Object Transaction Service, Version 1.1*, May 2000.

[OC96] Oudshoorn, M. J.; Crawley, S. C.: "Beyond Ada 95: The Addition of Persistence and its Consequences", in *Reliable Software Technologies - Ada-Europe'96, Montreux, Switzerland, June 10-14, 1996*, pp. 342 – 356, Lecture Notes in Computer Science **1088**, Springer Verlag, 1996.

[PK84] Papadimitriou, C. H.; Kanellakis, P. C.: "On Concurrency Control by Multiple Versions", *ACM Transactions on Database Systems **9**(1)*, March 1984, pp. 89 – 99.

[PKH88] Pu, C.; Kaiser, G. E.; Hutchinson, N. C.: "Split-Transactions for Open-Ended Activities", in *14th International Conference on Very Large Data Bases*, pp. 26 – 37, Los Angeles, California, 1988, Morgan Kaufmann.

[PMJPA98] Patiño-Martinez, M.; Jiménez-Peris, R.; Arévalo, S.: "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada", in *Reliable Software Technologies - Ada-Europe'98, Uppsala, Sweden, June 8-12, 1998*, pp. 78 – 89, Lecture Notes in Computer Science **1411**, 1998.

[PS88] Parrington, G. D.; Shrivastava, S. K.: "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", in *2nd European Conference on Object–Oriented Programming (ECOOP '88)*, pp. 233 – 249, Berlin, August 1988, Lecture Notes in Computer Science **322**, Springer Verlag.

[PSWL95] Parrington, G. D.; Shrivastava, S. K.; Wheater, S. M.; Little, M. C.: "The Design and Implementation of Arjuna", in *Computing Systems*, volume 8, pp. 255 – 308, Berkeley, CA, USA, Summer 1995, USENIX.

[PW97] Pautet, L.; Wolf, T.: "Transparent Filtering of Streams in GLADE", in *TRI-Ada'97 Conference*, pp. 11 – 19, St. Louis, MO, November 1997, ACM Press.

[Ran75] Randell, B.: "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering **1**(2)*, 1975, pp. 220 – 232.

[RBP+91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[RC97] Ramamritham, K.; Chrysanthis, P. K.: “Advances in Concurrency Control and Transaction Processing”. Los Alamitos, California, 1997.

[RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G.: *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison Wesley Longman, Reading, MA, USA, 1999.

[RK01] Romanovksy, A.; Kienzle, J.: “Action-Oriented Exception Handling in Cooperative and Competitive Object-Oriented Systems”, in *Advances in Exception Handling Techniques*, Lecture Notes in Computer Science **2022**, Springer Verlag, 2001.

[Rom99] Romanovsky, A.: “On Structuring Cooperative and Competitive Concurrent Systems”, *The Computer Journal **42**(8)*, 1999, pp. 627 – 637.

[RSB+98] Riehle, D.; Siberski, W.; Bäumer, D.; Megert, D.; Züllighoven, H.: “Serializer”, In Martin et al. [MRB98], pp. 293 – 312.

[RX95] Randell, B.; Xu, J.: *The Evolution of the Recovery Block Concept*, chapter 1, pp. 1 – 21. in Lyu [Lyu95], 1995.

[RXR96] Romanovsky, A.; Xu, J.; Randell, B.: “Exception Handling and Resolution in Distributed Object-Oriented Systems”, in *ICDCS ’96; Proceedings of the 16th International Conference on Distributed Computing Systems; May 27-30, 1996, Hong Kong*, pp. 545 – 553, Washington - Brussels - Tokyo, May 1996, IEEE.

[SDP91] Shrivastava, S. K.; Dixon, G. N.; Parrington, G. D.: “An Overview of the Arjuna Distributed Programming System”, *IEEE Software **8**(1)*, January 1991, pp. 66 – 73.

[SD+85] Spector, A. Z.; Daniels, D. et al.: “Distributed Transactions for Reliable Systems”, in *Proceedings of the 10th ACM Sympsium on Operating System Principles*, pp. 127 – 146, Orcas Island WA, USA, December 1985, ACM SIGOPS Operating Systems Review **19**(5).

[SGR97] Strigini, L.; Giandomenico, F. D.; Romanovsky, A.: “Coordinated Backward Recovery between Client Processes and Data Servers”, *IEEE Proceedings - Software Engineering **144**(2)*, April 1997, pp. 134 – 146.

[SHM+00] Shannon, B.; Hapner, M.; Matena, V.; Davidson, J.; Pelegri-Llopart, E.; Cable, L.: *Java 2 Platform Enterprise Edition: Platform and Component Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 2000.

[Shr95] Shrivastava, S. K.: “Lessons Learned from Building and Using the Arjuna Distributed Programming System”, in *Theory and Practice in Distributed Systems*, pp. 17 – 32, Lecture Notes in Computer Science **938**, 1995.

[SMR93] Shrivastava, S. K.; Mancini, L. V.; Randell, B.: “The Duality of Fault-Tolerant System Structures”, *Software-Practice and Experience **23**(7)*, July 1993, pp. 773 – 798.

[SPB88] Spector, A. Z.; Pausch, R. F.; Bruell, G.: “Camelot: A Flexible, Distributed Transaction Processing System”, in *Proceedings of the 33rd IEEE Computer Society International Conference (Spring COMPCON 88)*, pp. 432 – 437, San Francisco CA, USA, March 1988, IEEE Computer Society Press.

[SPW+94] Shrivastava, S. K.; Parrington, G. D.; Wheater, S. M.; Little, M. C.; Caughey, S.; Ingham, D.; Calsavara, A.; Smith, J.: *Reliable Distributed Programming in C++: The Arjuna System Programmer’s Guide*, 1994.

[Str91] Stroustroup, B.: *The C++ Programming Language, Second Edition*. Addison–Wesley, Reading, MA, USA, 1991.

[Sun98] Sun Microsystems: *Java Object Serialization Specification*, November 1998.

[SW95] Stroud, R. J.; Wu, Z.: “Using Metaobject Protocols to Implement Atomic Data Types”, in *9th European Conference on Object–Oriented Programming (ECOOP ’95)*, pp. 168 – 189, August 7–15, 1995, Aarhus, Denmark, August 1995, Lecture Notes in Computer Science **952**, Springer Verlag.

[TLP+93] Thomsen, B.; Leth, L.; Prasad, S.; Kuo, T.-M.; Kramer, A.; Knabe, F. C.; Giacalone, A.: “Facile Antigua Release – Programming Guide”. *Technical Report ECRC-93-20*, European Computer Industry Research Centre, Munich, Germany, December 1993.

[TOM99] Tripathi, A.; Oosten, J. V.; Miller, R.: “Object-Oriented Concurrent Programming Languages and Systems”, *Journal of Object-Oriented Programming **12**(7)*, November / December 1999, pp. 22 – 29.

[Vac00] Vachon, J.: *COALA: A Design Language for Reliable Distributed Systems*. PhD Thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, December 2000.

[VCK96] Vlissides, J.; Coplien, J. O.; Kerth, N. L. (Eds.): *Patterns Languages of Program Design 2*. Addison Wesley, Reading, MA, USA, 1996.

[VR99] Vogel, A.; Rangarao, M.: *Programming with Enterprise JavaBeans, JTS, and OTS: Building Distributed Transactions with Java and C++*. John Wiley and Sons, New York, NY, USA, 1999.

[VRS86] Vinter, S.; Ramamritham, K.; Stemple, D.: "Recoverable Actions in Gutenberg", in *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 242 – 249, Los Angeles, Ca., USA, May 1986, IEEE Computer Society Press.

[Weg90] Wegner, P.: "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger **1**(1)*, August 1990, pp. 7 –87, ACM.

[Wir88] Wirth, N.: "Type Extensions", *ACM Transactions on Programming Languages and Systems **10**(2)*, April 1988, pp. 204 – 214.

[WJS+00a] Wellings, A. J.; Johnson, B.; Sanden, B.; Kienzle, J.; Wolf, T.; Michell, S.: "Integrating Object-Oriented Programming and Protected Objects in Ada 95", *ACM Transactions on Programming Languages and Systems **22**(3)*, May 2000, pp. 506 – 539, ACM Press.

[WJS+00b] Wellings, A. J.; Johnson, B.; Sanden, B.; Kienzle, J.; Wolf, T.; Michell, S.: "Object-Oriented Programming and Protected Objects in Ada 95", in *Reliable Software Technologies - Ada-Europe'2000, Potsdam, Germany, June 26-30, 2000*, pp. 16 – 28, Lecture Notes in Computer Science **1845**, 2000.

[WK99] Warmer, J.; Kleppe, A.: *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series, Addison–Wesley, Reading, MA, USA, 1999.

[WL85] Weihl, W. E.; Liskov, B.: "Implementation of Resilient, Atomic Data Types", *ACM Transactions on Programming Languages and Systems **7**(2)*, April 1985, pp. 244 – 269, ACM Press.

[WS99] Wolf, T.; Strohmeier, A.: "Fault Tolerance by Transparent Replication for Distributed Ada 95", in *Reliable Software Technologies - Ada-Europe'99, Santander, Spain, June 7-11, 1999*, pp. 411 – 424, Lecture Notes in Computer Science **1622**, 1999.

[XRR00] Xu, J.; Romanovsky, A.; Randell, B.: "Concurrent Exception Handling and Resolution in Distributed Object Systems", *IEEE Transactions on Parallel and Distributed Systems **11**(11)*, November 2000, pp. 1019 – 1032.

[XRR+95] Xu, J.; Randell, B.; Romanovsky, A.; Rubira, C. M. F.; Stroud, R. J.; Wu, Z.: "Fault Tolerance in Concurrent Object-Oriented Software through Coordi-

nated Error Recovery", in *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pp. 499 – 509, Pasadena, California, 1995.

[XRR+99] Xu, J.; Randell, B.; Romanovsky, A.; Stroud, R. J.; Zorzo, A. F.; Canver, E.; von Henke, F.: "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions", in *FTCS-29: 29th International Symposium on Fault Tolerant Computing*, pp. 68 – 75, Madison, USA, 1999.

Annex B: Author and Citation Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W

X

Y

Z

Curriculum Vitae

Jörg Andreas Kienzle, born on the 31st of July, 1970 in Princeton, New Jersey, USA.

Education

1989	Graduated from high school (Realgymnasium Basel, Switzerland), "Matura Typ B" (Latin bias).
1992 - 1997	Computer science studies at the Swiss Federal Institute of Technology (EPFL) in Lausanne, graduated 1997 with the Engineering Diploma.
1994 - 1995	Exchange year at Carnegie Mellon University, Pittsburgh, USA.

Research

1995 - 1996	Cryptology: Support for transparent data encryption in distributed Ada.
1996 - 1997	Distributed Systems: Support for multi-client / server applications in distributed Ada. Diploma thesis title: "Network Application Support for Ada 95".
1997 - 2001	Ph.D. thesis in the field of transaction systems.

Teaching

1997 - 1998	Main assistant of the course "CASE Tools and Project Management", given to 4th year computer science students.
1998 - 1999	Assistant responsible for the implementation phase of the software engineering project for 3rd year computer science students.
1999 - 2000	Main assistant for the course "Programming I & II" for 1st year computer science students, given in English.
2000 - 2001	Lecturer of the course "Programming I & II" for 1st year computer science students, given in English.

Other Activities

1989 - 1992	Figure skating training in Füssen and Oberstorf, Germany. Swiss Ice Dancing Champion in 1992, 18th / 27 at the 1992 World Figure Skating Championships, San Francisco, California, USA.

Publications

1996: J. Kienzle, T. Wolf, and A. Strohmeier: “Secure Communication in Distributed Ada”, in *Reliable Software Technologies - Ada-Europe’96, Montreux, Switzerland, June 10-14, 1996*, pp. 198 – 210, Lecture Notes in Computer Science **1088**, Springer Verlag, 1996.

1997: J. Kienzle: “Network Applications in Ada 95”, in *TRI-Ada’97 Conference*, pp. 3 – 9, St. Louis, MO, November 1997, ACM Press.

1999: J. Kienzle and A. Strohmeier: “Shared Recoverable Objects”, in *Reliable Software Technologies - Ada-Europe’99, Santander, Spain, June 7-11, 1999*, volume 1622 of *Lecture Notes in Computer Science*, pp. 397 – 411, 1999.

J. Kienzle: “Combining Tasking and Transactions”, in *Proceedings of the 9th International Real-Time Ada Workshop, Wakulla Springs Lodge, Tallahassee FL, USA, March 1999*, pp. 49 – 53, Ada Letters XIX(2), June 1999, ACM Press, 1999.

2000: A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell: “Integrating Object-Oriented Programming and Protected Objects in Ada 95”, *ACM Transactions on Programming Languages and Systems **22**(3)*, May 2000, pp. 506 – 539, ACM Press.

J. Kienzle, A. Romanovsky, and A. Strohmeier: “A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages”. *Technical Report EPFL-DI No 2000/335*, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

J. Kienzle: “Exception Handling in Open Multithreaded Transactions”, in *ECOOP Workshop on Exception Handling in Object-Oriented Systems, Cannes, France*, June 2000.

J. Kienzle and A. Romanovsky: “On Persistent and Reliable Streaming in Ada”, in *Reliable Software Technologies - Ada-Europe’2000, Potsdam, Germany, June 26-30, 2000*, pp. 82 – 95, Lecture Notes in Computer Science **1845**, 2000.

J. Kienzle and A. Romanovsky: “Combining Tasking and Transactions, Part II: Open Multithreaded Transactions”, in *Proceedings of the 10th International Real-Time Ada Workshop, Castillo de Magalia, Las Navas del Marqués, Avila, Spain, September 2000*, pp. 67 – 74, Ada Letters XXI(1), ACM Press, March 2001.

A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell: “Object-Oriented Programming and Protected Objects in Ada 95”, in *Reliable Software Technologies - Ada-Europe’2000, Potsdam, Germany, June 26-30, 2000*, pp. 16 – 28, Lecture Notes in Computer Science **1845**, 2000.

A. Romanovksy and J. Kienzle: Action-Oriented Exception Handling in Cooperative and Competitive Object-Oriented Systems", in *Advances in Exception Handling Techniques*, Lecture Notes in Computer Science **2022**, Springer Verlag, 2001.

2001: J. Kienzle, A. Romanovsky, and A. Strohmeier: "Open Multithreaded Transactions: Keeping Threads and Exceptions under Control", in *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Roma, Italy, 8 - 10 January, 2001*, 2001. To be published.

J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martinez: "Transaction Support for Ada", in *Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, May 14-18, 2001*, pp. 290 – 304, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.

X. Caron, J. Kienzle, and A. Strohmeier: "Object-Oriented Stable Storage based on Mirroring", in *Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, May 14-18, 2001*, pp. 278 – 289, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.